(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

PERSISTENT SEARCH:
A BRIDGE BETWEEN DEPTH-FIRST AND
BREADTH-FIRST SEARCH FOR PHYSICAL AGENTS

by

Michael McClanahan Mayer

June 1989

Thesis Advisor: Man-Tak Shing

Approved for public release; distribution unlimited

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: UNCLASSIFIED	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (If Applicable): 52	7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (city, state, and ZIP code): Monterey, CA 93943-5000	7b Address (city, state, and ZIP code): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number
8c Address (city, state, and ZIP code)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

11 Title (Include Security Classification)
PERSISTENT SEARCH: A BRIDGE BETWEEN DEPTH-FIRST AND BREADTH-FIRST SEARCH FOR PHYSICAL AGENTS

12 Personal Author(s) Mayer, Michael McClanahan

13a Type of Report	13b Time Covered	14 Date of Report (year, month,day)	15 Page Count
Master's Thesis	From To	1989 June	76

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes		
Field	Group	Subgroup

18 Subject Terms (continue on reverse if necessary and identify by block number)
Artificial Intelligence; Search; Path Planning; Complexity Analysis;

19 Abstract (continue on reverse if necessary and identify by block number)
Current search algorithms and heuristics perform very poorly in the highly realistic scenario of a physical agent traversing an initially unknown search space. They do not attempt to minimize the amount of movement required by the physical agent attempting to reach a desired goal location. In order to overcome the failings of these algorithms in dealing with searches of this particular nature, a new algorithm called persistent search was created.

Persistent search differs from most other algorithms because it focuses on minimizing the physical movement of an active agent traversing an unknown search space, coping with the physical aspects of the problem which are too often ignored. Persistent search uses several standard search techniques but applies them in such a way as to change the semantics of the search. An interesting additional property of this algorithm is that through the manipulation of a single control variable, termed the persistence factor, the operation of the basic algorithm can be changed to span the continuum of behaviors between depth-first and breadth-first search.

20 Distribution/Availability of Abstract	21 Abstract Security Classification
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	Unclassified

22a Name of Responsible Individual	22b Telephone (Include Area code)	22c Office Symbol
Prof. Man-Tak Shing	(408) 646-2634	Code 52Sh

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page
Unclassified

Approved for public release; distribution is unlimited.

Persistent Search:
A Bridge Between Depth-first and Breadth-first Search For Physical Agents

by

Michael McClanahan Mayer
Lieutenant, United States Navy
B.A., Northwestern University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

Author: _____
Michael McClanahan Mayer

Approved By: _____
Man-Tak Shing, Thesis Advisor

_____
Gordon H. Bradley, Second Reader

_____
Robert B. McGhee, Chairman, Department of Computer Science

_____
Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

Current search algorithms and heuristics perform very poorly in the highly realistic scenario of a physical agent traversing an initially unknown search space. They do not attempt to minimize the amount of movement required by the physical agent attempting to reach a desired goal location. In order to overcome the failings of these algorithms in dealing with searches of this particular nature, a new algorithm called *persistent search* was created.

Persistent search differs from most other algorithms because it focuses on minimizing the physical movement of an active agent traversing an unknown search space, coping with the physical aspects of the problem which are too often ignored. Persistent search uses several standard search techniques but applies them in such a way as to change the semantics of the search. An interesting additional property of this algorithm is that through the manipulation of a single control variable, termed the persistence factor, the operation of the basic algorithm can be changed to span the continuum of behaviors between depth-first and breadth-first search.

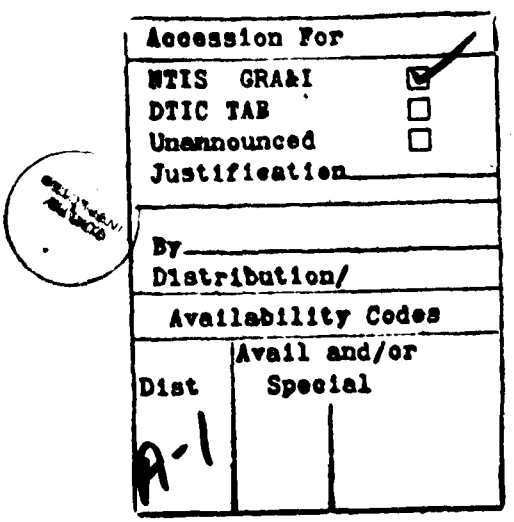


TABLE OF CONTENTS

I. INTRODUCTION 1

II. PROBLEM STATEMENT 4

III. SURVEY OF CLASSIC SEARCH METHODS 9

A. DEPTH-FIRST SEARCH 9

B. BREADTH-FIRST SEARCH 10

C. EVALUATION AND COST FUNCTIONS 10

D. HILL-CLIMBING SEARCH 12

E. BEST-FIRST SEARCH 12

F. BRANCH AND BOUND 13

G. A* SEARCH 14

H. PATH PLANNING VS. PATH FINDING 14

I. SUMMARY 15

IV. ALGORITHM DESCRIPTION AND IMPLEMENTATION 16

A. GENERAL ALGORITHM 16

B. SEARCH OVER A RECTILINEAR MAZE 19

V. ANALYSIS OF ALGORITHM AND SEARCH COMPLEXITY 23

A. SEARCH COMPLEXITY 23

B. ALGORITHM COMPLEXITY 27

C. COMPLEXITY SHORTCUTS 29

VI. EMPIRICAL TESTING AND RESULTS 31

A. MAZE GENERATION 31

B. HILL-CLIMBING SEARCH EVALUATION 33

VII. FUTURE RESEARCH 39

APPENDIX A PERSISTENT SEARCH C CODE 42
APPENDIX B PERSISTENT SEARCH LISP CODE 53
APPENDIX C EMPIRICAL TESTING C CODE 58
APPENDIX D RESULTS OF SAMPLE TEST CASE 65
LIST OF REFERENCES 67
BIBLIOGRAPHY 68
INITIAL DISTRIBUTION LIST 69

I. INTRODUCTION

Current search algorithms and heuristics perform very poorly in the highly realistic scenario of a physical agent traversing an initially unknown search space. They do not attempt to minimize the amount of movement required by the physical agent to reach a desired goal location. Most algorithms ignore the physical aspects of search, measuring the quality of their solutions only by the amount of computation required. In order to overcome the failings of these search algorithms in dealing with searches of this particular nature, a new algorithm called *persistent search* was created and is presented here.

Persistent search differs from most other algorithms because it focuses on minimizing the physical movement of an active agent traversing an unknown search space rather than on the number of comparisons. To deal with this twist on a classic search problem, persistent search copes with the physical aspects of the problem which are too often ignored.

The difference between persistent search and other standard search algorithms likens to the difference between internal and external sorting algorithms. External sorting algorithms use the same methods as internal sorting algorithms but apply these methods differently. External sorting algorithms take into account the fact that they must interact with a mechanical device which is inherently very slow in comparison to strictly electronic operations.

Persistent search uses several standard search techniques but applies them in such a way as to change the dynamics of the search. Persistent search provides the agent with the flexibility to handle the physical constraints of the problem. In a search over unknown terrain, where the relative location of the goal is known but the feasible paths to the goal are

unknown, the search involves not only movement to the goal but also exploration. Exploration is the process of learning about one's surrounding environment.

Since the agent does not start with a map, it can only learn about its environment by moving through it. If a map were available to the agent, standard path planning algorithms could do relatively inexpensive computation to find the optimum path (shortest, least cost, most efficient, etc.) to the goal before the first step was ever taken by the agent. Without a map, the agent must instead physically roam its environment to learn about it by sensing the immediate surroundings. While exploration is movement which adds to the agent's knowledge of its environment, a different kind of physical movement, backtracking, does not provide this offsetting benefit.

Backtracking in the physical sense, unlike backtracking in classic search methods, entails the physical movement of the agent through its environment. Backtracking is especially costly for two reasons. First, it entails physical movement to make a transition from one state to another in the search space and such movement is vastly slower than computation. Second, it implies movement which does not add to the agent's knowledge of the search space. In most classic search algorithms, no cost is associated with backtracking: none is needed. A pointer is merely redirected or a context is popped off a stack. In contrast, persistent search associates a cost with backtracking which enables it to minimize physical movement in a realistic way.

Persistent search is the embodiment of a new methodology for treating search by a physical agent which trades additional computation for a reduction in physical movement. An interesting additional property of this algorithm is that through the manipulation of a single control variable, termed the persistence factor, the operation of the basic algorithm

can be changed to span the continuum of behaviors between depth-first search and breadth-first search.

II. PROBLEM STATEMENT

A search problem may be characterized by a start state, a description of a set of goal states and a set of transition operators which transform one state into another. A state is a collection of information of sufficient content to uniquely differentiate one state from another. If there exists a transition operator which transforms a state ***A*** into a state ***B*** then state ***B*** is called the immediate successor of state ***A***. The set of all states reachable from the start state through the application of transition operators is known as the search space.

A search problem may be represented by a graph and in special cases by a tree rooted at the start state as shown in Figure 2.1. Search has the effect of overlaying a tree of explored nodes onto the search space even when the search space may not naturally have a tree-like structure. The depth of the search is the length of the longest path to any state in this tree. A search space is naturally represented by a tree if there exists no transitions which create circuits. Circuits are transitions which allow a state to be visited more than once along a path in the graph. Circuits create connections between separate branches of the graph.

Since persistent search operates on a search space represented by a graph, it can therefore be considered as a variation of the graph traversal problem. A graph consists of a finite set of nodes ***V*** joined by a set of undirected arcs ***E***. The nodes represent states in the search space while the arcs represent legal transitions between states.

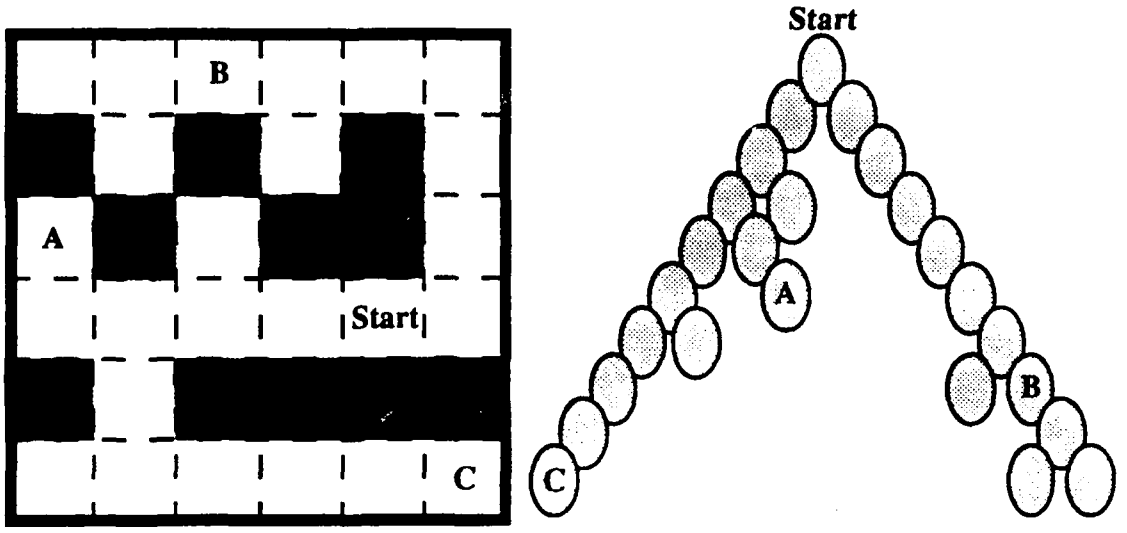


Figure 2.1 Maze with Tree Representation

The terms node, state, and location all represent the same concept of situation or condition, but each implies a different context. Node is used when discussing search in the context of graph traversal. Use of the term state implies a general or abstract discussion of search. When referring to states as locations, the context is that of the chosen problem domain, search in a rectilinear maze.

Traversing the graph is a single physical agent which attempts to arrive at a single distinguished node called the goal state. The agent, which has no prior knowledge of the graph, attempts to minimize the amount of physical movement expended searching for the goal. Although the agent has no knowledge of any paths to the goal, it does have a method for judging its nearness to the goal. This nearness can be measured in a number of ways including absolute distance, number of arcs to the goal, amount of difference between the description of the agent's current state and the goal state, etc.

The identity of the physical agent performing the search is not specified in the problem. It could be an autonomous vehicle or robot attempting to find its way across

unfamiliar terrain, able only to sense its immediate surroundings. The agent could also be a packet of information traversing a packet switching network from the node of the packet's sender to the node of its addressee. The routing logic at each packet switching node knows only which nodes are attached to it, not the entire set up of the network.

In a more restrictive case, the agent could be a read/write head on a tape drive. Here the environment (tape) moves instead of the agent (read/write head), but the effect is the same. In optimum search on a tape (Hu, 1987, pp. 573-590), ***n*** records are stored alphabetically on a tape. A binary search of the tape minimizes the number of comparisons but not the amount of movement of the tape. A linear search of the tape minimizes the number of movements but not the number of comparisons. A tape-optimal search minimizes the total cost of comparisons and movements. Optimum search on a tape is considered a restricted case of the general problem because the search space can be naturally represented by a tree.

For the purpose of using a real world example, the case of a robot traversing a random maze is chosen and will be used throughout this thesis. Among all of the real world examples, it is the most directly applicable and simplest to grasp while still encompassing the full dynamics of the general problem.

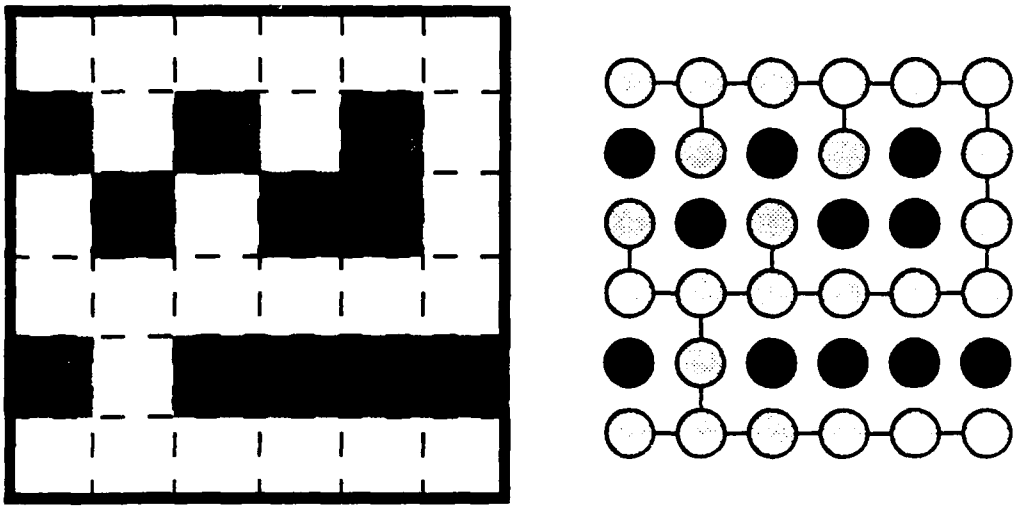


Figure 2.2 Maze with Rectilinear Graph Representation

The unknown terrain over which the robot will travel is represented by a rectilinear graph or grid-graph as shown in Figure 2.2. Nodes are arranged in a rectangular fashion with arcs connecting nodes immediately above and below and to the right and left (north, south, east and west respectively). Each node then has four possible immediate successors which are distinguished from one another by their x and y coordinates.

Locations within the maze are either passable or impassable. Impassable locations in the maze are represented by disconnected nodes in the graph, making them unreachable from any direction.

As stated above, the robot begins with no knowledge of the graph. It may learn about the graph only as it moves from node to node via the edges connecting them. The robot's ability to sense its surroundings is limited to determining whether the immediate successors of the current state are passable or impassable. In effect, the robot can judge whether it can move to the north, south, east or west. Since the terrain is a grid-graph, the robot is only able to move and survey in the four cardinal directions.

Additionally, the robot knows its position relative to the goal. Several different means could be used to accomplish this. If the robot initially knows the goal's position relative to its own, it can maintain this knowledge through dead reckoning as it moves about the maze from its initial starting point. Using this information the robot can determine its rectilinear distance from the goal as well as the distance from the goal to any immediate successor state. The following formula computes rectilinear distance from the node's x and y coordinates:

$$| X_r - X_g | + | Y_r - Y_g | \tag{2.1}$$

The r subscript is used to indicate the robot's coordinates while the g subscript indicates the goal's.

Using a different method, either an active or passive guidance system could also be used to direct the robot to the goal (e.g., either a form of radar or a beaconing system). While only lines of bearing are required to choose which immediate successor is closer to the goal, an exact location for the goal is required to totally order the search space. If the goal is northeast of the current location, then the states to the north and the east are closer to the goal and have the same rectilinear distance to the goal. However, when comparing widely separated locations, a ranking for the locations based on their closeness to the goal cannot be determined without an exact location for the goal. If location P is north of the goal and location Q is south of the goal, there is no static way to tell which one is closer merely by their direction to the goal. By triangulating lines of bearing taken from different locations, an exact location for the goal can be determined.

III. SURVEY OF CLASSIC SEARCH METHODS

Different search methods can be classified based on the way transition operators are applied to control the traversal of the search space. There are two major categories of search methods: depth-first and breadth-first.

A. DEPTH-FIRST SEARCH

Depth-first search is the simpler of the two. Beginning at the start state, depth-first search makes a transition to an immediate successor state. It then performs the same operation again moving to a successor state of that state. This continues until a goal state is reached or a state which has no successors is reached. If the current state has no successors, depth-first search backtracks along its path and tries to reach new states by taking previously unchosen transitions in the order in which they are reached while backtracking. One can also view depth-first search in the following way. At each state reached, depth-first search places all immediate successor states on a stack, a last-in, first-out (LIFO) data structure. It then pops a state off the stack and makes it the current state. If a state which has no successors is the current state, no transitions are pushed onto the stack. A state previously pushed onto the stack is then popped off and made the current state. A stack is a data structure which automatically supports this type of backtracking.

Depth-first search moves away from the start state very quickly but can pass by the goal, performing needless additional search. This becomes a very serious problem, especially when the search space is very large.

B. BREADTH-FIRST SEARCH

While in the current state, breadth-first search chooses every possible transition. It applies them all in turn, adding the immediate successor states reached by each transition to an agenda of new states not yet explored. States on the agenda are also called frontier states. Frontier states are states which have been examined and perhaps rated by either a cost or evaluation function, but not yet explored. A state has been explored if its immediate successor states have been examined and added to the agenda.

Breadth-first search adds all immediate successor states of the start state to the agenda. It then begins working its way from the beginning to the end of the agenda, adding the successors of the states it explores to the end of the agenda. The agenda is equivalent to a queue data structure, first-in, first-out (FIFO). The search continues until either the goal is found or until the agenda is empty, signifying the exhaustion of the search space. Typically, breadth-first search progresses very slowly but is guaranteed to eventually reach the goal if the goal is located within the search space. If the number of successors of each state is very large, breadth-first bogs down very quickly.

C. EVALUATION AND COST FUNCTIONS

Each of the above methods may be guided by an evaluation function, a cost function or both. An evaluation function is a way of associating a number with a state which is a measure of the state's goodness in regard to it leading to the goal state. Traditionally, the closer the value of the state's evaluation function is to zero, the higher the state's rank. An evaluation function always looks forward from the current state to the goal state and therefore its value is often referred to as a state's estimated future cost.

A cost function applies a price to a state transition or to an entire path. The value of the cost function tells how expensive a particular transition or path is. Again, as with evaluation functions, smaller is better — the lower the cost of a transition, the higher its ranking. A cost function measures from either the start state or current state to a successor state, associating the cost of the whole path with the successor state.

A convenient way to tell the difference between a cost and evaluation function is that a cost function looks at the states where the search has been while an evaluation function looks ahead and assesses where the search is going (Rowe, 1988, pg. 201). Some search techniques combine the cost and evaluation function in guiding the search. In such cases it is best to choose functions which are measured in the same units, since it often does not make sense to mix unrelated quantities (e.g., combining gallons of fuel expended with minutes of exposure to enemy fire). The sum of the values of a state's cost and evaluation functions is referred to as the state's *combined cost*.

To further clarify the use of cost and evaluation functions, their role in solving two problems will be examined: a bin packing and a maze problem. In the case of searching through a maze, every intersection in the maze is considered to be a state. The paths leading to and from the intersection are transitions leading to other states. The goal and start states are randomly chosen intersections.

A good evaluation function for this problem is a state's straight line distance to the goal state. If the agent's movement through the maze is limited to movement left, right, forward and back, then the state's rectilinear distance to the goal location should be the evaluation function. Both of the above functions have the desirable property that their return values are continuous and smooth. The values do not jump around wildly as the search progresses from state to state.

In a search of a maze with variable cost regions, where the cost of moving from one location to another is not constant, a cost function can be used to constrain the search. This is often done when searching for the least cost path. Only the move which would result in the least cost path from the start state to the new state is ever taken.

In the case of a bin packing problem, where a set of objects are to be stored in the fewest number of bins, the number, total weight or volume of items left to store could serve as the evaluation function. The cost function for this problem can be represented by the number of bins used. A transition has a cost of one if it causes a new bin to be used, zero otherwise. The total number of bins used would be the cost of the path composed of the set of transitions chosen.

D. HILL-CLIMBING SEARCH

Variations of depth-first and breadth-first search are possible through the use of cost and evaluation functions. Using an evaluation function with depth-first search, an immediate successor state which is closest to the goal will always be explored first. This is called hill-climbing. While this is very useful in guiding the search, it still very often leads to trouble. Because each transition decision is based solely on the current state, hill-climbing can very quickly go wrong due to local minimums. These are paths that start out looking promising but do not lead to the goal. They either end in a dead end or eventually turn and lead away from the goal.

E. BEST-FIRST SEARCH

A variation of a breadth-first search using an evaluation function which avoids the above problems is called best-first search. Best-first search adds the successors of the

current state to its agenda which is ordered by the value of each state's evaluation function. The state with the lowest value for its evaluation function is at the front of the agenda and is made the next current state. Its successors are then added to the agenda in order. This kind of agenda can be easily represented by the use of a priority queue or a min-heap.

Using best-first search, exploration can often expand towards the goal just as quickly as with hill-climbing search. Best-first search also has the added benefit of always moving to the state with the lowest evaluation function result, thus keeping the agent from searching at a depth beyond the depth of the goal, missing it just because it was initially misled by a local minimum.

F. BRANCH-AND-BOUND

Branch-and-bound (B&B) search methods use a cost function to constrain the search and may be either of a depth-first or breadth-first variety. Breadth-first B&B adds states to the agenda according to their cost function result. The lower the cost function, the higher on the agenda the state is placed. The state with the lowest cost function is explored next. Breadth-first B&B guarantees the first path to the goal found is the optimal or least cost path (Kumar, 1987, pp. 1000-1004). Of all search strategies, breadth-first B&B hops around the most since it is controlled by a cost function stemming from the start state. In an homogeneous cost search space, breadth-first B&B radiates symmetrically from the start state.

Depth-first B&B always moves to the immediate successor state which has the least cost. Depth-first B&B follows the path of least resistance and so is like a stream of water running down hill. Only when it is dammed up does it back up and start flowing down another branch.

G. A* SEARCH

A* (pronounced "A star") is a very powerful search method which combines a cost and evaluation function to produce a flexible and capable search method. A* adds states to its ordered agenda according to the combined cost of each state's evaluation and cost functions. It then explores the state at the front of the agenda. A*'s cost function measures from the start state to the state to be explored. Just like breadth-first B&B, A* guarantees the first path to the goal found is the optimal path under the extra condition that the evaluation function result for the current state is always less than or equal to the actual cost of the path from that state to the goal (Hart, 1968, pp. 100-107). A* is most often used in path planning problems.

H. PATH PLANNING VS. PATH FINDING

Path finding algorithms differ from path planning ones in that path planning algorithms require complete knowledge of the search space. They are used when it is important to find and record a particular path to the goal (usually the optimum). Path planning is a batch process that takes place prior to traversal of the search space by a physical agent. If the terrain can be represented as a graph without negative costs associated with the edges, then a variation of Dijkstra's algorithm (breadth-first branch & bound) can be used to very efficiently find the optimal path from the start state to the goal state. More complex search strategies, such as A*, are used when it is necessary to reduce the branching factor of the states and hence reduce the search space.

Path finding on the other hand is an interactive process that takes place as the graph is being traversed. No prior knowledge of the graph is assumed although the parts of the

graph which have been explored may be remembered. Unlike path planning, an optimum path seldom results from this kind of search. Any path found is considered a success. Path finding is only useful in those cases where incomplete information about the search space is available or full processing of the information is too costly. The use of path finding in the latter case is often called lazy evaluation.

I. SUMMARY

Each of the above algorithms may be the best search method for a particular set of circumstances, but each has significant weaknesses when dealing with the search problem of a physical agent traversing an unknown maze. In the next chapter, the persistent search algorithm will be described in detail, focusing on how it differs from the above methods and why it is particularly well suited to solve the selected search problem.

IV. ALGORITHM DESCRIPTION AND IMPLEMENTATION

The algorithm for persistent search will be described in two ways. The first description will be very general, suitable for application to any kind of search space. A very detailed description of the algorithm will then follow which is specific to the chosen problem domain of a physical search agent in a rectilinear maze.

A. GENERAL ALGORITHM

The algorithm for persistent search is presented in terms of a general search space with a cost and evaluation function. The algorithm is as follows:

Step 0: Add the start state to the list of frontier states and make it the current state.

Step 1: If the current state is the goal state, quit.

Step 2: If the current state is not the goal state, remove it from the list of frontier states.

Step 3: Examine all immediate successor states which have not yet been frontier states and add them to the list of frontier states.

Step 4: Examine the list of frontier states. If the list is empty, quit; the search has failed. If the list is not empty, traverse to the best frontier state on the list; make it the current state and then go to Step 1.

The best frontier state as described above in Step 4 is defined as the state, v, which minimizes the equation:

$$f(v) = g(v) * pf + h(v) \tag{4.1}$$

where $g(v)$ is the cost of traversing to the state v from the current state; $h(v)$ is a lower bound estimate for the estimated future cost, the cost of traversing from the state v to the goal; and pf is the persistence factor, a coefficient for discounting the cost of backtracking.

While the above algorithm is very close to the algorithm for A* search, one should note the subtle but important differences. $g(v)$ is computed from the current state to the frontier state, rather than from the start state. This has the effect of totally negating the impact of past movements on future decisions. This is desirable since for a physical agent, once a move has been made there is no taking it back (without exerting a like amount of effort). All that matters in the search is where the agent is now and where it is going, not where it has been.

A second difference shows in the use of the persistence factor as a cost coefficient for $g(v)$. The persistence factor varies between 0.0 and 1.0 and serves to discount the cost of backtracking versus estimated future cost, $h(v)$. By varying the persistence factor, the behavior of persistent can be dramatically altered. When the persistence factor is 0.0, the cost of backtracking from one state to another becomes zero, negating it. Hence the formula for rating frontier states reduces to:

$$f(v) = h(v) \tag{4.2}$$

which is equivalent to that used for best-first search. Each frontier node is ranked only according to its estimated future cost. The physical agent will move about the search space without regard for the amount of movement required, traversing to which ever state is closest to goal.

When the persistence factor is 1.0, the behavior of persistent search is equivalent to that of hill-climbing. Since the traversal cost from one state to another is fully counted, $f(v)$ for a immediate successor of the current state will always be less than that of any other

frontier state. This can most easily be seen when the cost for moving to an immediate successor is the same for all states and $h(v)$ is transitive. If $g(v)$ is one for an immediate successor then the $f(v)$ of the successor equals at most $h(v) + 1$. This will always be less than the combined cost of any other state since even if the $h(v)$ of another state is lower, the traversal cost to that state will more than offset it. If another frontier state's estimated future cost is $h'(v)$ and this is less than the successor state's $h(v)$, then they must be at least $|h(v) - h'(v)|$ away from each other. Figure 4.1 demonstrates this very clearly. State P's estimated future cost (or distance from the goal) is four. The estimated future cost of state Q is six. Because the current state is only one away from state Q, its total cost will always be lower than P's. This is true for all immediate successor states.

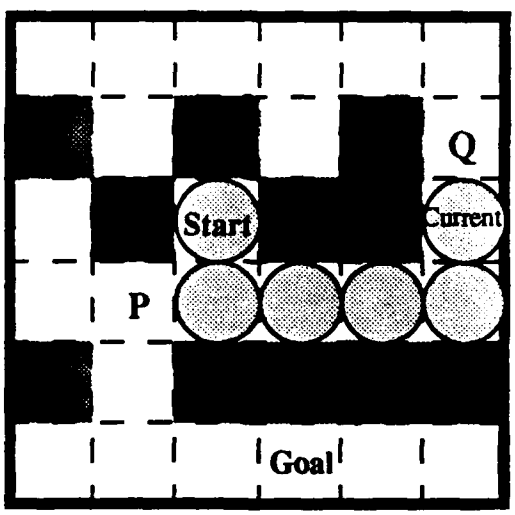


Figure 4.1 Maze with Frontier State P vs. Frontier State Q

Since an immediate successor state will always rank higher than any other frontier node when the persistence factor is 1.0, persistent search will never backtrack until it must. It only backtracks when a state has no legal successors. Again, it can easily be seen that

this equivalent to hill-climbing and depth-first search in general. When backtracking, the frontier node closest to the current state will be chosen. This happens for the same reasons as above, because the cost of traversal to a frontier state will always overcome any difference in estimated future cost among states.

B. SEARCH OVER A RECTILINEAR MAZE

The algorithm for persistent search which deals with the chosen problem domain of a rectilinear maze executes in two stages, with the second stage's execution depending on the results of the first stage. Several other changes in the algorithm have also been made for the sake of efficiency. Appendix A lists the source code for this specific implementation which will be explained in further detail below. The algorithm for persistent search in the context of an unknown maze is as follows:

Step 0: Add the start state to the list of frontier states and make it the current state.

Step 1: If the current state is the goal state, quit.

Step 2: If the current state is not the goal state, remove it from the list of frontier states.

Step 3: Examine all immediate successor states which have not yet been frontier states and add them to the list of frontier states.

Step 4: If any of the immediate successor states have a lower estimated future cost than the current state, move to that state, make it the current state and go to Step 1.

Step 5: If none of the immediate successor states has a lower estimated future cost than the current state, examine the list of frontier states.

If the list is empty, quit; the search has failed. If the list is not empty, traverse to the best frontier state on the list; make it the current state and then go to Step 1.

The best state is again defined as the state, v, which minimizes the equation:

$$f(v) = g(v) * pf + h(v) \tag{4.1}$$

The above algorithm accurately reflects a robot traversing a rectilinear maze trying to reach a goal location. Step 3 implements the first half of the persistent search algorithm, a hill-climbing search. The robot is able to evaluate immediate successor locations with respect to their distance from the goal. A successor state is made the current state if its rectilinear distance to the goal is less than the current state's. A simple north, east, south, west preference serves as a heuristic to break ties between successors of equal estimated future cost. This heuristic only results in multiple equivalent solutions when more than one path to the goal exists, such as when the maze is not a tree.

Step 3 greatly reduces the computation required by persistent search since the agenda does not need to be examined when a successor state which is closer to the goal exists. The reasoning for this is exactly the same as that stated above for why persistent search with a persistence factor of one acts like hill-climbing search. As long as the search is moving towards the goal no state on the list of frontier nodes has a lower combined cost than the successor state picked. The function **search()** in Appendix A implements this portion of the algorithm.

When no terrain map is available and a physical agent is performing the search, depth-first search variants such as hill-climbing and depth-first branch-and-bound are the only reasonable alternatives to persistent search. Breadth-first algorithms such as best-first

and breadth-first branch-and-bound jump around the search space excessively and perform too much unforced backtracking to be considered practical for a physical agent.

Best-first search does perform well, doing very little backtracking, if there are very few local minimums in the maze. Best-first search immediately backtracks if there is a state on the agenda with a lower estimated future cost than any of the successors of the current state. This happens when local minimums occur within the maze. As stated above, best-first search is equivalent to persistent search with a persistence factor of zero.

When there exists no successor to the current state which has a lower estimated future cost than the current state, the list of frontier states must be examined. This occurs in the function **getbestnode**() of the persistent search implementation. A min-heap maintains the list of frontier states with the state which has the lowest estimated future cost on top. Two-way links between the elements of the min-heap and the mapped portion of the maze are maintained to allow random access to the states on the min-heap and access to the state's description.

A breadth-first search of the known maze scans for the frontier state with the lowest combined cost each time the list of frontier states must be examined. This search is called a backtracking search since it determines which state the agent will move to next and the agent may have to backtrack through previously traversed states to reach the frontier state. The depth of the backtracking search represents the result of the cost function, $g(v)$, from equation (4.1) and is the traversal or shortest path cost to that state. The value of $g(v)$ is multiplied by the persistence factor to attain a modified traversal cost.

The estimated future cost of every frontier state reached by the backtracking search is summed with its modified traversal cost to give a combined cost for the state. During the course of the search, the state found with the lowest combined cost so far is saved. This

combined cost is then compared against a lower bound for the combined cost of the frontier state on top of the min-heap. Recall that the state on top of the min-heap is the frontier state with the lowest estimated future cost. The current depth is used as a lower bound for the traversal cost of this minimum frontier state. As the depth increases so does the lower bound of the traversal cost.

The backtracking search for the best frontier state ends when the combined cost of a frontier state matches or exceeds the lower bound combined cost of the state with the lowest estimated future cost. This check is made whenever the lowest combined cost state found changes or the depth changes. The backtracking search is terminated at the earliest possible point while still ensuring that the state with the lowest combined cost is the state to be explored next.

The state with the lowest combined cost represents the state most likely to lead to the goal without incurring overly expensive backtracking costs. The lower the persistence factor is for a search, the farther an agent will be allowed to travel to reach a state with a low estimated future cost. The persistence factor greatly influences the amount of backtracking which takes place during a search.

The path followed to every state during the backtracking search of the known maze is temporarily saved in each state's description so that once the best next state is found, the agent can traverse the indicated path to the state.

V. ANALYSIS OF ALGORITHM AND SEARCH COMPLEXITY

Formal complexity analysis of an algorithm determines the amount of resources (i.e., memory, time, etc.) needed by the algorithm as a function of the size of the problem instance (Brassard, 1988, pg. 5). The search complexity of a search problem is equal to the number of states evaluated. The algorithmic complexity of persistent search can then be stated as a function of the search complexity and the cost of making a transition during the search. The size of the search space places an upper bound on the search complexity, since the search space is the set of all reachable states. For an arbitrarily large search space, this result is not very satisfying since its limit grows without restriction.

A. SEARCH COMPLEXITY

A tighter bound for search complexity can be achieved by reasoning about the branching factor and the depth of a search problem (Rowe, 1988, pp. 207-208). The number of immediate successors of a state is called the branching factor of the state and usually includes only those successor states not already explored. If the branching factor for all states is not the same, then the average branching factor over all states is taken to be the branching factor for the search. The branching factor can also often be computed as the ratio of the number of states at depth $k + 1$ away from the start state to the number of states at depth k.

The lower bound for the depth of a search is the number of transitions along the shortest path in the search space from the start state to the goal space. The search complexity can then be stated as:

$$(B^{K+1} - 1) / (B - 1) \qquad (5.1)$$

where B is the average branching factor and K is this lower bound for the depth of the search (Rowe, 1988, pg. 208). For large branching factors, the equation (5.1) can be approximated by:

$$B^K \tag{5.2}$$

By convention, the term n shall often be substituted for equation (5.1), when referring to the size of a problem instance.

For the general case of graph traversal, the branching factor has an upper limit of V - 1, where V is the number of nodes in a completely connected graph. This then makes the depth of the search one, since every node is an immediate successor of every other node. The branching factor has a lower limit of one since the graph can be a chain of V nodes. In this instance, the depth has an upper limit of V - 1 since the start state and goal state could be on opposite ends of the linear chain. As can be seen for the general graph case, the size of the search space is bound by the depth and branching factor of the search.

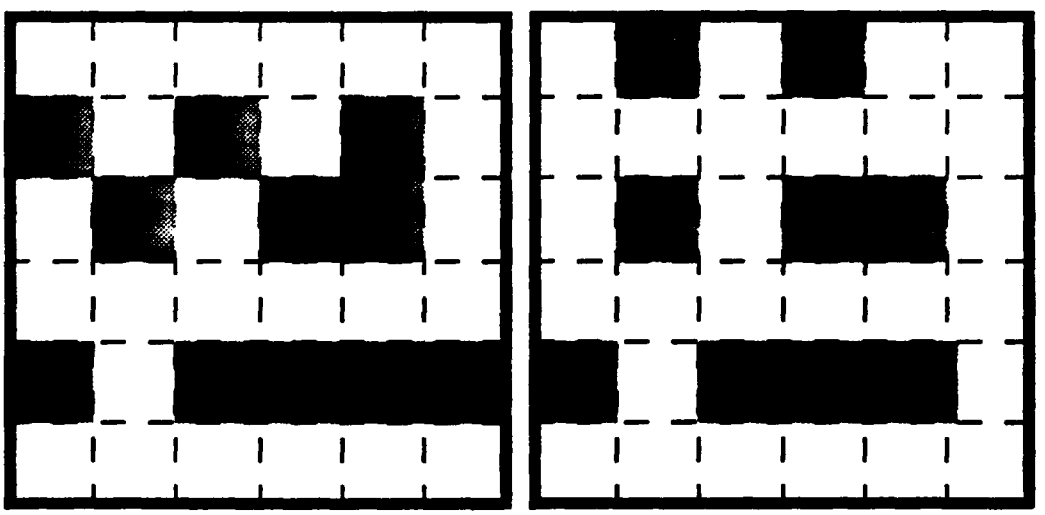


Figure 5.1 (a) Maze Without Cycles (b) Maze With Cycles

For the case of a rectilinear maze, an exact limit for the average branching factor can be computed in terms of the depth, K. A rectilinear maze with and without cycles is shown in Figures 5.1(a) and (b). Again, the main difference between the two types of mazes is in how they may be most naturally represented. As it happens, they also have different branching factors.

In the case of a rectilinear maze with cycles, the branching factor of the search is again severely constrained by the topology of the maze. At each deeper level of the search, the number of nodes on that level exceeds that of the previous level by four. This is illustrated in Figure 5.3. To calculate the upper bound on the number of nodes at any depth in the search of a rectilinear maze, merely multiply the depth by four.

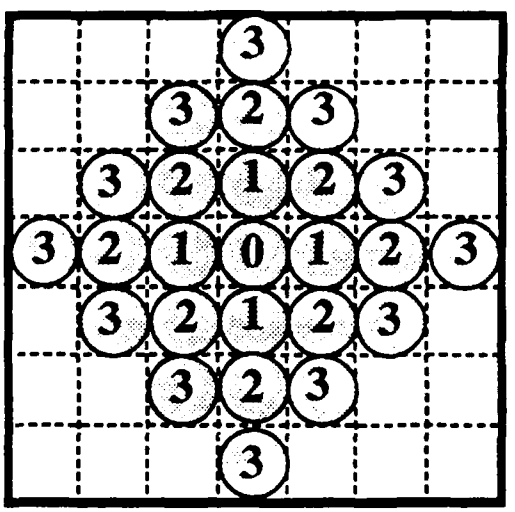


Figure 5.3 Maze With Cycles Showing Search Order

The size of a rectilinear search space with circuits is therefore bounded by a function of the depth of the search. For a rectilinear maze with circuits this is:

$$n = 2 * k^2 + 2 * k + 1 \tag{5.3}$$

where k is the current depth of the search and n is the number of states in the search tree.

For a search of sufficient depth over a maze without cycles, the limit of the average branching factor approaches one even faster than above. A single starting node in a complete rectilinear maze is allowed to expand freely and only those nodes which would result in circuits are disconnected. Figure 5.2 displays this scenario, showing expansion level by level. Expansion is constricted to the expansion of only four states at regular intervals, just as at a depth of three in the figure. The branching factor is severely constrained by the topology of the maze.

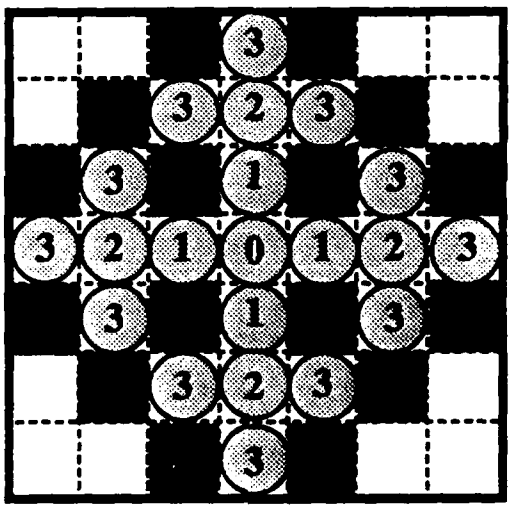


Figure 5.2 Maze Without Cycles Showing Search Order

For a rectilinear maze without circuits, the upper bound on the size of its search space is conjectured to be:

$$n = k^2 + 2 * k + 2 \tag{5.4}$$

B. ALGORITHM COMPLEXITY

Now that a tighter bound has been placed on the search complexity, $O(k^2)$, a reasonable worst-case complexity for the algorithm may be shown. By examining each step in the general algorithm for persistent search as stated in Chapter IV, a computational cost in terms of n may calculated. Step 0 is only done once and takes constant time. Steps 1 through 4 may be done up to n times, once for each state in the search space. No state is ever examined more than once. The complexity of any one of these steps must therefore be multiplied by n.

In Step 1, the current state is checked to see if it is the goal state. The amount of computation needed for this is proportional to the amount of information needed to distinguish one node for another. In the strictest sense, this increases as the $\log_2$ of the number of nodes, n. This is most easily shown if each node is uniquely identified by an integer. The more nodes, the greater the number of bits needed to represent the integer signifying a particular node. So the complexity of Step 1 is $O(\log_2 n)$ when including bit complexity. Bit complexity is often ignored when determining algorithm complexity thus making the time complexity of Step 1 constant, or $O(1)$.

In Step 2, the current state is removed from the list of frontier states. Locating the current state in the agenda list can be done in constant time if a link to the state's position in the list is kept in the state's representation. In the implementation of persistent search discussed above, the frontier list is kept in a min-heap so as to allow easy access to the smallest element and efficient updating of the list. Removing an element from the min-heap requires $O(\log_2 n)$ time.

All immediate successors of the current state which have never been frontier states are examined and added to the list of frontier states in Step 3. The complexity of finding

immediate successors of the current state is equivalent to that of the transition operators multiplied by the number of successor states. The operators are assumed to take constant time and since any state in a rectilinear graph has at most four immediate successors, the complexity is merely a constant multiplied a constant. Telling whether a state has been a frontier state also takes constant time since that information is part of its state description. Adding the states to the frontier list takes $O(\log_2 n)$ time for a min-heap.

Step 4 examines the list of frontier states and traverses to the best one. This takes $O(n)$ time since every state in the search space up to that point in the search may have to be examined to determine the best state. Traversing to the best state on the list also make take up to $O(n)$ time in the worst-case.

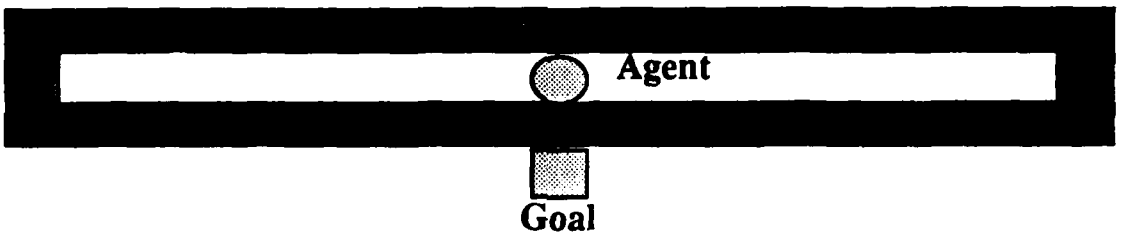


Figure 5.4 Worst-case Maze for Persistent Search

Figure 5.4 shows an example of a worst-case situation. The maze is linear with the goal unreachable. Persistent search will constantly have to explore the entire search space to determine the best state, no matter what the persistence factor. Several methods for combatting this are discussed in the next section of this chapter.

By combining the time complexities for the above steps, the exact time complexity for persistent search is given by the following equation:

$$T(n) = c1 + n * (c2 + c3 * \log_2 n + c4 * \log_2 n + c5 * n) \qquad (5.5)$$

$c1$ through $c5$ are constant factors. $c1$ represents the time complexity of Step 0 and the terms inside the parentheses represent the time complexity for Steps 1 through 4.

For sufficiently large n, only the highest order term is significant; the rest can be dropped. Ignoring lower order terms, Equation (5.5) simplifies to:

$$T(n) = c * n^2 \tag{5.6}$$

where c is a constant. Substituting Equation (5.3) for n in Equation (5.6)

$$T(n) = c * (2 * k^2 + 2 * k + 1)^2 \tag{5.7}$$

the computational complexity of the algorithm is shown in terms of k. If the depth of the search is limited to k, then ignoring lower order terms the limit is:

$$T(k) = c * k^4 \tag{5.8}$$

While the time complexity of persistent search is of $O(k^4)$, or $O(n^2)$, its space complexity is of $O(n)$. Only enough memory to store the state information and the frontier list is needed and both of these items are of $O(n)$.

C. COMPLEXITY SHORTCUTS

Several methods for reducing the complexity of persistent search are possible under certain conditions. With a persistence factor of 1.0, there exists no chance of backtracking before a state without successors is reached. There is no reason then for evaluating any state beyond the first frontier state found when backtracking from the current state. Setting the persistence factor to n ensures that the first frontier state reached will be have a lower combined cost at the time it is reached than the frontier state on the top of the min-heap. This results in the backtracking search terminating immediately and with the agent traversing to the first frontier state found. By limiting the backtracking search in this way, the amount of computation performed by persistent search is reduced to within a constant

factor of the amount of computation performed by hill-climbing search. Since n is normally not known prior to the search, a very large constant can be used for the persistence factor.

By maintaining the min-heap in the implementaion, persistent search is able to cut off the search for the best frontier state at the earliest opportunity, after it is sure it has found the best state. Although worst-case scenarios can still be created which require $O(n^2)$ time, the average case complexity is much lower. Empirical testing of the algorithm has shown very favorable results in comparison to hill-climbing search.

VI. EMPIRICAL TESTING AND RESULTS

In order to better judge the performance of the persistent search algorithm, a random maze generator and a hill-climbing algorithm were both implemented. See Appendix C to examine the actual code. These implementations were not only useful in testing persistent search but also in improving it. The testing also revealed several weaknesses in the testing methodology which was subsequently revised.

A. MAZE GENERATION

The random maze generator operates in two modes, creating mazes with or without cycles. The mazes without cycles can be naturally represented as trees and those with cycles can be represented as graphs. This mode implementation was necessary because of the nature of persistent search's backtracking. When backtracking, persistent search always follows the shortest path to the next frontier state. This gave persistent search an extra advantage over hill-climbing search which can only trace back along its original path to its next frontier state. In non-tree mazes, this results in a large amount of back and forth movement by the hill-climbing algorithm as it winds its way out of large open areas. This inefficient behavior can be seen clearly in Figure 6.1. Persistent search cuts across its own path following short circuits in the terrain. If the maze is a tree, there are no circuits, which offsets persistent search's advantage and allows it to be more accurately evaluated.

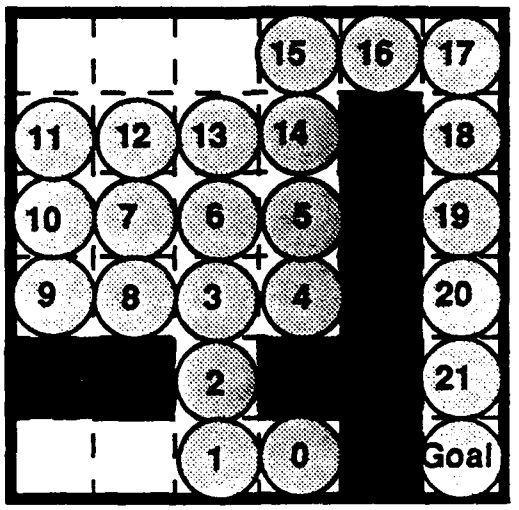


Figure 6.1 Hill-climbing Search in a Maze With Cycles

The function **makemaze**() implements the maze generation algorithm. Using a depth-first algorithm to follow a random path through an empty maze, **makemaze**() creates obstacles at intervals along its path according to a maze density parameter. **makemaze**() begins by creating the bounds for the maze and then picks a random starting point within the maze. The algorithm maintains a trail of the locations in the maze it has explored.

At each step along its path, it randomly determines whether it must turn from its current direction. If so, it places an obstacle in the direction it was going and continues on in a random new direction. The higher the maze density, the more likely it is that the path will turn and an obstacle will be created. If the maze generator becomes blocked by its own trail and obstacles, it backs along its trail until it comes to an open location to which it then moves.

If **makemaze**() is restricted to creating mazes which are trees, it must also check before it takes a step to see if the next location it is going to explore will create a cycle in the

maze. If so, it places an obstacle there instead and chooses a new direction. In a maze without circuits, there is only one path from any state to any other state in the maze.

B. HILL-CLIMBING SEARCH EVALUATION

For comparison with persistent search, a hill-climbing algorithm was implemented. It is very similar to the algorithm used to create the maze as they are both depth-first algorithms. An evaluation function determining which successor location was nearer the goal was added to the maze generating depth-first algorithm to create the function **dfs**().

dfs() uses a simple stack to keep its current path and a bitmap to mark where it has been. It places all successor states on the stack and then moves to the state on the top of the stack. **dfs**() uses the same heuristic that persistent search does. It chooses between successor states with equal estimated future costs by using a simple north, east, south, and west preference. Again, **dfs**() performs exactly like persistent search with a persistence factor of one, choosing to explore the same nodes in the same order.

The results comparing hill-climbing to persistent search are very favorable for the new algorithm. With very few exceptions, a persistence factor for the search can be found which beats hill-climbing search and minimizes the amount of movement required by the agent to find it way to the goal. In all cases, persistent search can do at least as well as hill-climbing search.

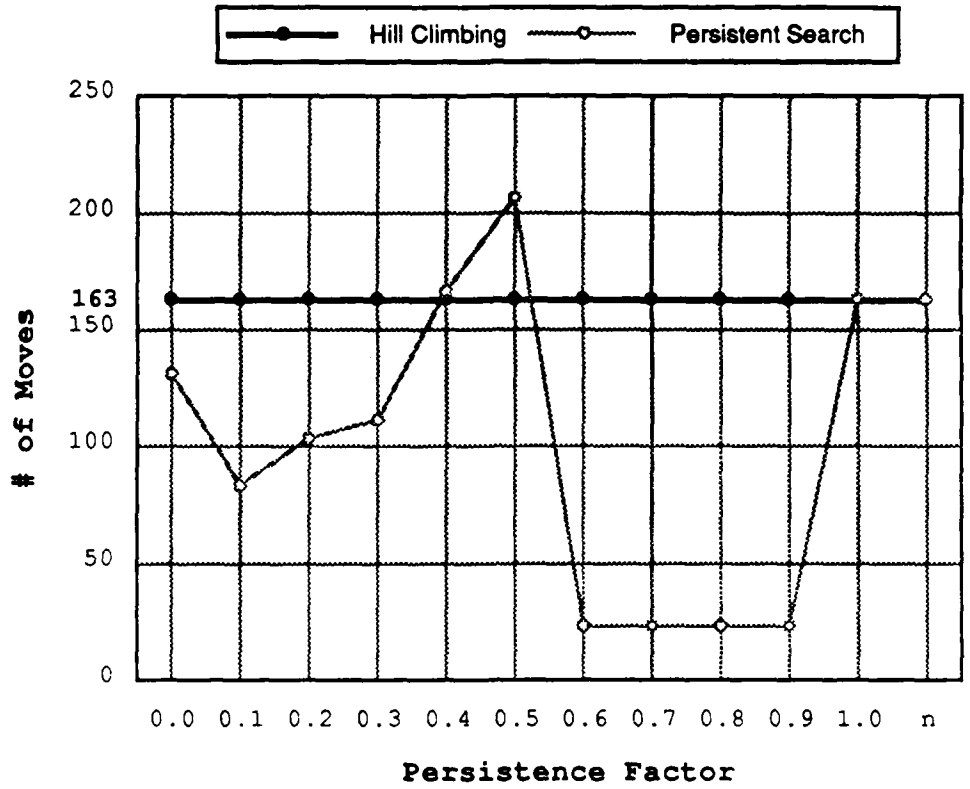


Figure 6.2 — Graph of moves vs. persistence factor for a maze of size 16 X 16 and a maze density of 0.5

Both the number of moves made by the agent and the amount of computation required were recorded for each test run. The graphs in Figures 6.2 through 6.5 show the number of moves and the amount of computation for two representative test runs and a range of persistence factors.

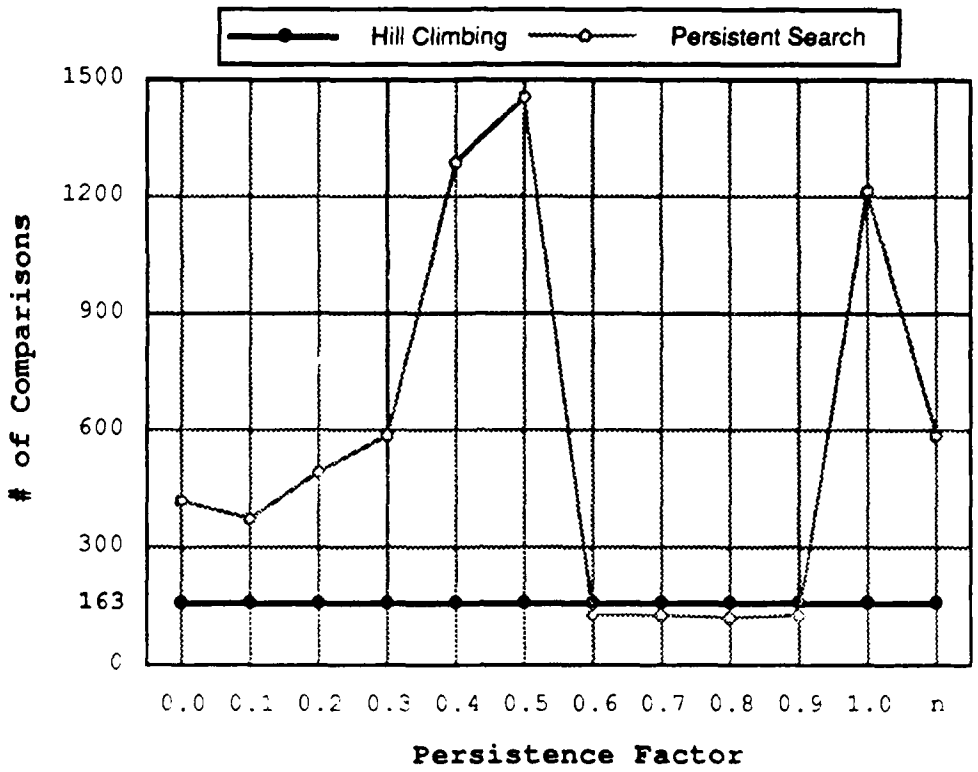


Figure 6.3 — Graph of computation vs. persistence factor for a maze of size 16 X 16 and a maze density of 0.5

A persistence factor of ***n*** applies only to reducing the amount of computation. The persistence factor is set to a number larger than the number of states in the search space. This reduces backtracking search and makes the amount of computation done by persistent search equivalent (within a constant factor) to that done by hill-climbing search.

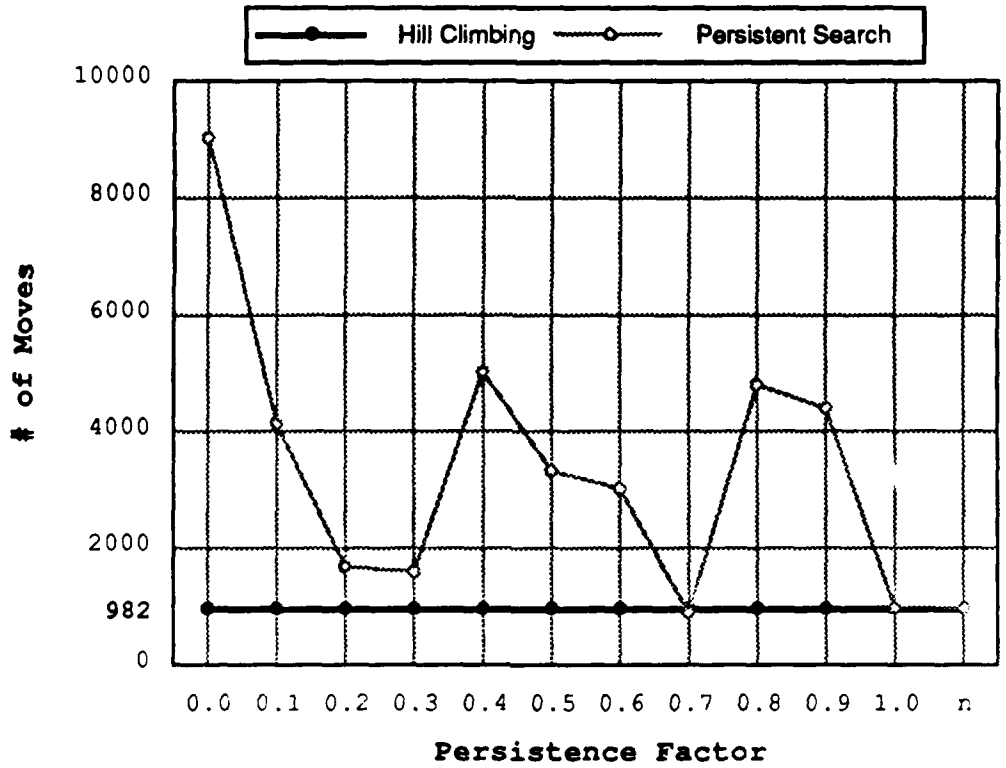


Figure 6.4 — Graph of moves vs. persistence factor for a maze of size 64 X 64 and a maze density of 0.5

As can be seen from the graphs, both the number of moves and the amount of computation vary greatly with the persistence factor. There is no smooth curve, because persistent search depends on catching discontinuities in the graph. Persistent search excels when a local minimum is followed which quickly turns away from the goal. Hill-climbing must follow these local minimums to their end while persistent search can cut them short and move to more promising paths.

Over 27,000 test mazes were created and comparisons of persistent search and hill-climbing search performed. On randomly generated mazes which were trees, persistent search performed better, making fewer moves, than hill-climbing search over 40% of the mazes with some persistence factor and performed at least as well with some persistence

factor in 100% of the cases. In mazes which were not trees, persistent search bettered hill-climbing in 87% of the test runs with some persistence factor.

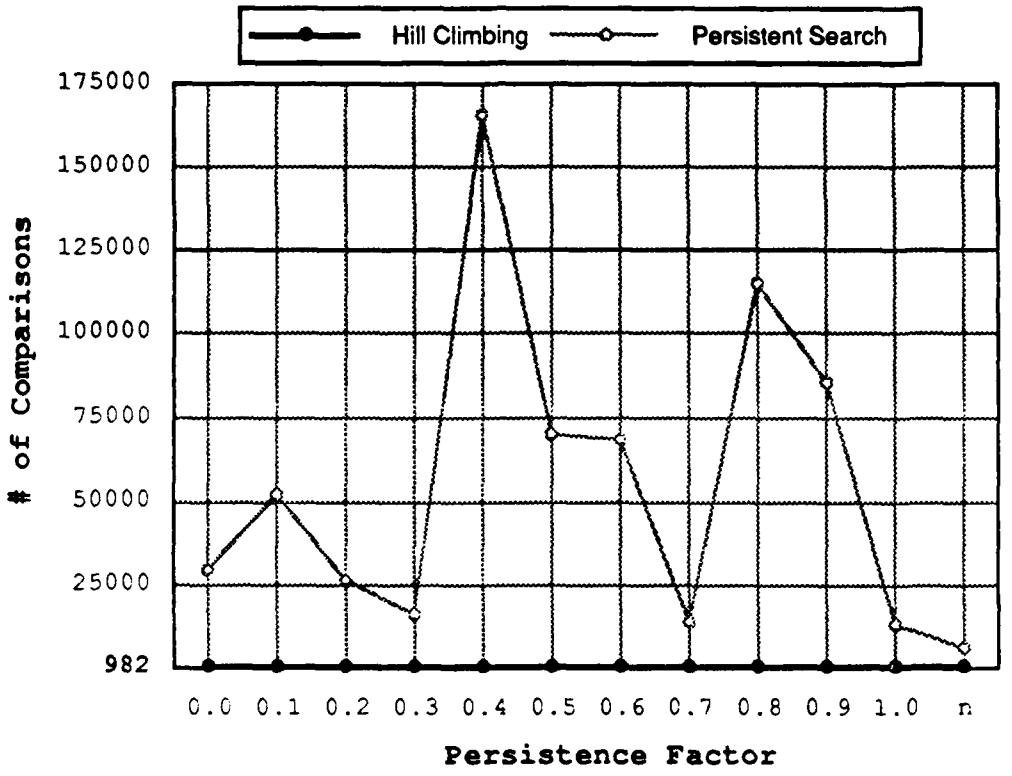


Figure 6.5 — Graph of computation vs. persistence factor for a maze of size 64 X 64 and a maze density of 0.5

Several factors influenced the results of these tests in the hill-climbing algorithm's favor. First, the search space is bounded. Hill-climbing can stray from the path to the goal and wander forever while following a local minimum and never reach the goal unless the search space is bounded or an arbitrary limit for the depth of the search is set. Persistent search needs no such boundary as it uses its persistence factor to cut short a path but then possibly later continuing that path. Bounding the search space has the effect of setting an

arbitrary limit for the depth of a search without the possibility of cutting short the search too soon.

A second advantage is that the random maze is generated with an algorithm very similar to **dfs()**. Both are depth-first searches in nature, but **makemaze()** uses a random heuristic to choose the next state while **dfs()** uses an evaluation function. This advantage was curtailed by using a new random starting location for the search than the one used by **makemaze()** to create the maze.

As can be seen in the results for the number of computations performed by persistent search, higher time complexity algorithms soon outstrip constants and lower order algorithms no matter how large their coefficients. This implies there is a limit to the size of the search space that persistent search can traverse before the computation time for the search takes longer than the time needed to physically traverse the search space. The time needed to physically traverse the search space is proportional to the number of states. The time complexity of persistent search is proportional to the square of the number of states.

In the broadest sense, persistent search applies to any search where physical constraints apply to the exploration of the search space. This applies to movement by a physical agent, be it a robot or a read/write head on a mass storage device, as well as to the transmission time of a network packet of information.

VII. FUTURE RESEARCH

A number of possible areas of future research suggest themselves from the empirical testing and conclusions. The first and the closest to the heart of this research is the implementation of methods for changing the persistence factor in an adaptive manner during the course of a search. The second is the use of persistence search in non-homogeneous cost search spaces, consisting of non-unit cost transitions. The first area of research provides ways to improve the performance of the algorithm, the second expands the set of problems to which it can be applied.

Although persistent search always performs as well as hill-climbing, it is very dependent on the choice of the right persistence factor. Even so, search spaces are not completely consistent in that a single persistence factor operates best throughout the course of a search. When the search space is a tree, there exists only one path to the goal. The job of persistent search is to see that the agent does not stray too far from that path. There are multiple "fateful turns" along that path When the persistence factor is too high and a wrong turn is made, no correction of the error is possible before the agent is whisked away. When the persistence factor is too low, the agent is plagued by indecision, running back and forth over its path unable to move forward in a definitive way.

Without global knowledge of the search space, no perfect method exists for adjusting the persistence factor during the course of a search, although several useful heuristics suggest themselves. The first heuristic for improving the performance of persistent search involves regulating the persistence factor in inverse proportion to the density of the maze. Empirical testing revealed a tendency for a search with a high persistence factor to do better on a less dense maze. In the corresponding opposite case, a search with a lower

persistence factor does better in a more dense maze. This trend was very strong when testing on non-tree mazes, but was inconclusive when testing on tree mazes. One factor in this is that density is far less distinct when also preserving the tree property of the maze.

The above results tend to bear out one's natural intuition. When the terrain is wide open and there are but a few small obstacles, it pays to just go directly around them rather than double back and try a different tack. The search is very unlikely to be led too far out of it way. When the terrain is dense and large obstacles are common, then dead ends and local minimums are common. It pays to do some backtracking in order to find a path to the goal.

Another heuristic for adjusting the persistence factor would be to vary it with the distance from the goal state. When the current state is far from the goal, make the agent more persistent so that it will cover a larger area with less backtracking. When the agent is close to the goal, have it be less persistent so that it will cover the terrain surrounding the goal more thoroughly. The author calls this method the bird dog technique. When a bird dog is on the trail of a pheasant, let's say, and the scent is faint (the pheasant is far away), the bird dog very quickly covers a large amount of terrain in an ambling manner. When the scent is strong, the dog's movement and attention becomes very concentrated. The dog traces and retraces its path covering a very small area very thoroughly. Judging from hunting dogs observed in the past, this method is very effective.

An area of research which would greatly expand the applicability of persistent search would be to implement it for a search problem where the costs of making a transition from state to state is variable. This research would dovetail well with current research within the Computer Science Department at the Naval Postgraduate School. It could easily be added

to a number of ongoing projects. The change to the algorithm itself is slight and only a single variable need be added to each state description.

To handle variable cost transitions, add a variable to each state description to hold that state's path length from the current state. Now when performing the search over the known terrain for the best next state, instead of merely keeping track of the depth of the search, update each state with its path length from the current state. Use the state's path length instead of depth to calculate the best next state. As one can easily see, the changes required to handle this new expanded problem domain are almost trivial.

Persistent search is totally new in its perspective on the problem of search and much more work will be required to fully ascertain its worth. Many new applications and deeper insights await to be discovered.

APPENDIX A

PERSISTENT SEARCH C CODE

```
/*****************************************************************************
*
*  psearch.c -     A program which demonstrates and evaluates the persistent
*                  search algorithm as specified in the NPS Master's Thesis
*                  titled: Persistent Search: A Bridge Between Depth-first and
*                  Breadth-first Search For Physical Agents, June 1989.
*
*  Written by:     Michael M. Mayer, LT, USN
*  Date:           29 May 1989
*  Version:        1.1
*
*  Compiler:       Written in C on a VAX 11/785.  Versions also available for
*                  the Apple Macintosh and Silicon Graphics IRIS4D/70GT
*  Options:        The following compiler options are allowed via the -D option
*    TREE:         Causes makemaze() to produce mazes which are trees rather
*                  than more general graphs.
*    PM:           Causes the display of the mazes generated by makemaze()
*    PP:           Causes the algorithms to print their path as they search
*
*  Compiling:      cc psearch.c [-DTREE] [-DPM] [-DPP] -o psearch
*  Usage:          psearch <maze size> <maze density> <persistence factor>
*
*****************************************************************************/

    /* include files */
#include <stdio.h>

#define FALSE           0
#define TRUE            1
        /*  max frontier nodes = 2x + 2, where x = path length. Therefore
            n = 2x + 2 + x, where n = # nodes */
#define MAXFRNTRNODES   (size * size / 3 * 2 + 2)
#define NUMLINKS        4    /* number of connected nodes (NESW) */
#define UNKNOWN         0    /* code for node never before seen */
#define VISITED         1    /* code for node that has been traversed */
#define FRONTIER        2    /* code for node that is on the frontier -
                                seen but not traversed */
    /* recti-linear distance */
#define dist(a,b,c,d)   (abs((a) - (c)) + abs((b) - (d)))

    /* turn x,y coordinate into array index */
#define indx(x,y)       ((y) * size + (x))

    /* get x coordinate from array index */
#define getx(x)             ((x) % size)

    /* get y coordinate from array index */
#define gety(y)             ((y) / size)
```

```
typedef struct a_node
{
    unsigned int

            estfutcost,       /* estimated future cost */
                              /* result of the evaluation function */
                              /* node's rectilinear distance to the goal */
            parent,           /* index of node used to get to this node
                                 during backtracking search*/
            links[NUMLINKS],  /* links to the North, East, South, West */
            status,           /* whether the node has not been seen before,
                                 visited, or on the frontier */
            mhindex,          /* index of a frontier node on the minheap */
            searchid;         /* index of node started from during
                                 search for best frontier node
                                 guarenteed to be unique since a node is
                                 never searched from twice */
} node;

unsigned char     *world;     /* a bitmap holding the representation
                                 of the terrain.  In a robot, this bitmap
                                 supplies input to the robot's sensors */
unsigned char     *trail;     /* a bitmap representing the nodes traversed
                                 by the makemaze function */
unsigned int      *pstack;    /* a stack holding the path being followed by
                                 the makemaze function */
node              *myworld;   /* the two dimensional array of nodes which
                                 holds the robot's representation of the
                                 terrain seen so far */
unsigned int      *heap;      /* a minheap which holds the frontier nodes
                                 in order of their estimated future cost */
unsigned int      *agenda;    /* a circular queue holding the nodes seen
                                 but not yet expanded in a breadth-first
                                 which attempts to find the best frontier
                                 node to go to next */

unsigned int      compares,   /* number of operations performed
                                 by persistent search */
                  moves,      /* number of moves by the agent traversing
                                 the maze from the start to the goal using
                                 persistent search */
                  dfscompares, /* number of operations performed
                                 by depth-first search (hill-climbing) */
                  dfsmoves;   /* number of moves by the agent traversing
                                 the maze from the start to the goal using
                                 depth-first search (hill-climbing) */

unsigned int      size,       /* size on a side of the maze, a square
                                 size X size large */
                  Start,      /* the index of the start node in the maze */
                  goal;       /* the index of the goal node in the maze */

double            pf,         /* persistance factor of the search */
                  density;    /* density of the maze */
```

```
main(argc, argv)
int argc;
char **argv;
{
     unsigned int  x, y;          /* loop variables */
     double        atof();        /* ascii to floating point library function */

          /* check number of arguments passed to psearch */
     if (argc < 4)
     {
          fprintf(stderr, "Error: too few arguments.\n");
          exit(1);
     }

          /* get parameters:     1) size of the maze
                                 2) density of the maze
                                 3) persistence factor of the search */
     size = atoi(argv[1]) / 8 * 8;
     density = atof(argv[2]);
     pf = atof(argv[3]);

          /* allocate dynamic memory for data structures */
     world = (unsigned char *)calloc(size * size / (sizeof(char) * 8),
          sizeof(char));
     trail = (unsigned char *)calloc(size * size / (sizeof(char) * 8),
          sizeof(char));
     pstack = (unsigned int *)calloc(size * size, sizeof(unsigned int));
     myworld = (node *)calloc(size * size, sizeof(node));
     heap = (unsigned int *)calloc(MAXFRNTRNODES + 1, sizeof(int));
     agenda = (unsigned int *)calloc(MAXFRNTRNODES + 1, sizeof(int));

          /* create random maze */
     makemaze();
          /* print out maze */
     print_maze();
          /* perform depth-first (hill-climbing) search on maze */
     dfs();
          /* perform persistent search on maze */
     init_heap();
     addheap(Start);
     search(getx(Start), gety(Start), getx(goal), gety(goal));

          /* print results of search */
     printf("persistance factor:%.2f\nmaze density:%.2f\n", pf, density);
     printf("dfs:%4d ps:%d moves\n", dfsmoves, moves);
     printf("dfs:%4d ps:%d comparisons\n", dfscompares, compares);

          /* free dynamic memory */
     free(heap);
     free(agenda);
     free(myworld);
     free(world);
     free(pstack);
     free(trail);
}  /* main */
```

```
int search(x, y, goalx, goaly)
unsigned int  x, y,              /* coordinates of current node */
              goalx, goaly;      /* coordinates of goal node */
{
    int       i,                 /* link direction from current node */
              best;              /* index of best next node */

#ifdef PP     /* print out coordinates of current node */
    printf("PS visited: %d, %d\n", x, y);
#endif

    if (! atgoal(x, y, goalx, goaly))   /* see if the goal has been found */
    {
            /* keep statistics */
        moves++;
        compares++;

            /* update map and list of frontier nodes */
        alterheap(myworld[indx(x, y)].mhindex);
        myworld[indx(x, y)].status = VISITED;
        setlinks(x, y, goalx, goaly);

            /* find out if there exists a successor node which:
                1) is reachable from the current node
                2) has not yet been visited and
                3) is closer to the goal than the current node
        i = 0;
        while( i < NUMLINKS &&
            !(myworld[indx(x, y)].links[i] &&
            myworld[myworld[indx(x, y)].links[i]].status != VISITED &&
            closer(x, y, getx(myworld[indx(x, y)].links[i]),
                gety(myworld[indx(x, y)].links[i]), goalx, goaly)))
            i++;

            /* if such a node exists go there and continue search */
        if (i < NUMLINKS)
            search(getx(myworld[indx(x, y)].links[i]),
                    gety(myworld[indx(x, y)].links[i]), goalx, goaly);
        else
            /* if not, get the best node from the list of frontier nodes */
        {
            best = getbestnode(x, y);
            if (best >= 0)
                /* if get back a node index, move to that
                   node and continue search */
            {
                printsteps((unsigned int)best, indx(x, y));
                search(getx(best), gety(best), goalx, goaly);
            }
            else
                    /* if returns -1, list is empty and search fails */
                printf("PS: No solution possible!\n");
        }
    }  /* if */
}  /* search */
```

```
int getbestnode(x, y)
unsigned int x, y;              /* coordinates of current node */
{
    int     i,                  /* link direction */
            ci,                 /* current link index */
            minindex = -1,      /* index of node with lowest combined cost */
            depth = 0,          /* depth of backtracking search */
            numthislevel = 1,   /* number of nodes at cuurent depth */
            numnextlevel = 0,   /* number of nodes one level deeper */
            id  = indx(x, y),   /* unique search identifier */
            head = 0,           /* head of circular queue */
            tail = 0;           /* tail of circular queue */

                /* combined cost of node with lowest combined cost */
    float   mincost = (float)size * size * pf + size * size + 1,
            currcost;           /* combined cost of node being evaluated */

    agenda[tail++] = indx(x, y);    /* insert into circular queue */
    while (head != tail)            /* while circular queue not empty */
    {
        compares++;                 /* keep statistics */

            /* if through with all the nodes at the current depth */
        if (!numthislevel)
        {
            depth++;                        /* increase depth */
            numthislevel = numnextlevel;    /* get # of nodes at new depth */
            numnextlevel = 0;

                /* see if combined cost of best node found so far is better
                   than the lower bound for the combined cost of node with
                   the lowest estimated future cost */
            if (mincost <= (float)myworld[findmin()].estfutcost +
                           (float)depth * pf)
                break;   /* if so then search is complete */
        }
```

```
            if (myworld[agenda[head]].status == VISITED) /* not frontier node */
                for(i = 0; i < NUMLINKS; i++)            /* for each edge */
                {
                    ci = myworld[agenda[head]].links[i]; /* set current index */

                        /* if that node has not already been examined
                           during the backtracking search */
                    if (ci && myworld[ci].searchid != id)
                    {
                        myworld[ci].parent = agenda[head];  /* set parent */
                        myworld[ci].searchid = id;          /* mark search */
                        agenda[tail] = ci;                  /* add to queue */
                        tail = ++tail % MAXFRNTRNODES;      /* turn corner */
                        numnextlevel++;                 /* add to next level */
                    }
                }
            else     /* this is a frontier node */
            {
                    /* get combined cost of frontier node being examined */
                currcost = (float)myworld[agenda[head]].estfutcost +
                           (float)depth * pf;
                    /* if better than current minimum combined cost node */
                if (currcost < mincost)
                {
                    minindex = agenda[head];    /* get new mincost node */
                    mincost = currcost;         /* get new mincost */

                        /* check  combined cost just like the time above*/
                    if (mincost <= (float)myworld[findmin()].estfutcost +
                                   (float)depth * pf)
                        break;    /* if so then search is complete */
                }
            }
            numthislevel--;                 /* done checking another node */
            head = ++head % MAXFRNTRNODES;  /* turn corner of queue */
        }  /* while */
        return(minindex);       /* return index of lowest combined cost node */
    }  /* getbestnode */


    int closer(x1, y1, x2, y2, goalx, goaly)
    unsigned int  x1, y1,           /* coordinates of current node */
                  x2, y2,           /* coordinates of successor node */
                  goalx, goaly;     /* coordinates of goal node */

        /* if current node is closer to goal than successor node, return true */
    {
        return(dist(x1, y1, goalx, goaly) > dist(x2, y2, goalx, goaly));
    }  /* closer */
```

```
setlinks(x, y, goalx, goaly)
unsigned int  x, y,              /* coordinates of current node */
              goalx, goaly;      /* coordinates of goal node */
{
     /* get frontier nodes, fill in their state description, and add
        them to the list of frontier nodes */

     if(!getblk(x, y - 1, world))     /* North */
     {
          myworld[indx(x, y)].links[0] = indx(x, y - 1);
          if(!myworld[indx(x, y - 1)].status)          /* if status UNKNOWN */
          {
               myworld[indx(x, y - 1)].estfutcost =
                    dist(x, y - 1, goalx, goaly);
               myworld[indx(x, y - 1)].status = FRONTIER;
               addheap(indx(x, y - 1));
          }
     }
     if(!getblk(x + 1, y, world))     /* East */
     {
          myworld[indx(x, y)].links[1] = indx(x + 1, y);
          if(!myworld[indx(x + 1, y)].status)          /* if status UNKNOWN */
          {
               myworld[indx(x + 1, y)].estfutcost =
                    dist(x + 1, y, goalx, goaly);
               myworld[indx(x + 1, y)].status = FRONTIER;
               addheap(indx(x + 1, y));
          }
     }
     if(!getblk(x, y + 1, world))     /* South */
     {
          myworld[indx(x, y)].links[2] = indx(x, y + 1);
          if(!myworld[indx(x, y + 1)].status)          /* if status UNKNOWN */
          {
               myworld[indx(x, y + 1)].estfutcost =
                    dist(x, y + 1, goalx, goaly);
               myworld[indx(x, y + 1)].status = FRONTIER;
               addheap(indx(x, y + 1));
          }
     }
     if(!getblk(x - 1, y, world))     /* West */
     {
          myworld[indx(x, y)].links[3] = indx(x - 1, y);
          if(!myworld[indx(x - 1, y)].status)          /* if status UNKNOWN */
          {
               myworld[indx(x - 1, y)].estfutcost =
                    dist(x - 1, y, goalx, goaly);
               myworld[indx(x - 1, y)].status = FRONTIER;
               addheap(indx(x - 1, y));
          }
     }
}  /* setlinks */
```

```
init_heap()             /* initialize heap by setting its size to zero */
{
    heap[0] = 0;
}  /* init_heap */


int findmin()           /* return node with the lowest estimated future cost */
{
    return(heap[1]);
}


addheap(new)
unsigned int new;       /* item to be added to heap is index into myworld */
{
    heap[0]++;                              /* increment heap size */
    heap[heap[0]] = new;                    /* put index into last position */
    myworld[new].mhindex = heap[0];         /* set link into minheap */
    percolate(heap[0]);                     /* let item bubble up minheap */

}  /* addheap */


alterheap(rm)
unsigned int rm;        /* index of element in heap to be deleted */
{
    int oldval;

    if (rm > heap[0])       /* ***ERROR*** */
    {
        fprintf(stderr, "Tried removing element [%d] with heap size [%d]\n",
            rm, heap[0]);
        exit(1);
    }
    else if (rm == heap[0])     /* if removing last element in heap */
    {
        heap[0]--;              /* just decrement heap size */
        return;
    }
    else                        /* remove element and adjust heap */
    {
        oldval = myworld[heap[rm]].estfutcost;
        heap[rm] = heap[heap[0]];           /* replace element to be removed
                                               with last element */
        heap[0]--;                          /* just decrement heap size */
        myworld[heap[rm]].mhindex = rm;     /* adjust link to minheap */

            /* decide whether to bubble element up or down */
        if (myworld[heap[rm]].estfutcost < oldval)
            percolate(rm);
        else
            settle(rm);
    }
}  /* alterheap */
```

```
percolate(start)                    /* bubble element up in heap */
unsigned int start;                 /* starting position within heap */
{
     int i,                         /* heap index */
         temp;                      /* temporary swap variable */

     for (i = start; i != 1; i /= 2)     /* from start to every parent */
     {
          compares++;               /* keep statistics */
               /* if cost of child is more than the parent then swap them */
          if (myworld[heap[i]].estfutcost < myworld[heap[i / 2]].estfutcost)
          {
               temp = heap[i / 2];
               heap[i / 2] = heap[i];
               heap[i] = temp;
               temp = myworld[heap[i / 2]].mhindex;
               myworld[heap[i / 2]].mhindex = myworld[heap[i]].mhindex;
               myworld[heap[i]].mhindex = temp;
          }
          else
               break;   /* done bubbling up */
     }
}  /* percolate */

settle(start)                       /* bubble element down in heap */
unsigned int start;                 /* starting position within heap */
{
     int i,                         /* heap index */
         temp,                      /* temporary swap variable */
         child;                     /* index of child */

     i = start;                     /* current index is start */
     while (i <= heap[0] / 2)       /* while current index is not a leaf index */
     {
          compares++;               /* keep statistics */
          child = 2 * i;            /* get child index */
               /* if second child exists and its estimated future cost is less
                  than the first child's, make it the current child */
          if (child + 1 <= heap[0] && myworld[heap[child + 1]].estfutcost <
                                      myworld[heap[child]].estfutcost)
               child++;
               /* if cost of child is less than the parent then swap them */
          if (myworld[heap[i]].estfutcost > myworld[heap[child]].estfutcost)
          {
               temp = heap[child];
               heap[child] = heap[i];
               heap[i] = temp;
               temp = myworld[heap[child]].mhindex;
               myworld[heap[child]].mhindex = myworld[heap[i]].mhindex;
               myworld[heap[i]].mhindex = temp;
          }
          else
               break;   /* done bubbling down */
          i = child;    /* current element is now the child element */
     }  /* while */
}  /* settle */
```

```
int atgoal(x, y, goalx, goaly)
unsigned int  x, y,              /* coordinates of current node */
              goalx, goaly;      /* coordinates of goal node */
{
    return(x == goalx && y == goaly);   /* if at goal node return true */
}  /* atgoal */


printsteps(finish, start)
unsigned int  finish,            /* index of best next node */
              start;             /* index of current node */
{
    int currnode = myworld[finish].parent,
        old = finish,
        real_old = finish;

    while (currnode != start)  /* reverse pointers along path */
    {
        old = currnode;
        currnode = myworld[currnode].parent;
        myworld[old].parent = real_old;
        real_old = old;
    }
    currnode = old;
    while (currnode != finish)      /* while not at end of path */
    {
        moves++;                    /* keep statistics */

#ifdef PP                           /* print out traversed node */
        printf("PS visited: %d, %d\n", getx(currnode), gety(currnode));
#endif

        currnode = myworld[currnode].parent;     /* follow path */
    }
}  /* printsteps */
```

```
setblk(x, y, bitmap)
unsigned int  x, y;                    /* coordinates in bitmap */
unsigned char     *bitmap;             /* bitmap in which to set value */
/* sets the bit in the bitmap corresponding to the provided x,y coordinate */
{
     unsigned int  byte,               /* byte where x,y coord is located */
                   bit;                /* bit in byte where x,y is located */

     byte = indx(x, y) / (sizeof(char) * 8);
     bit = indx(x, y) % (sizeof(char) * 8);
     bitmap[byte] |= 1 << (7 - bit);
}  /* setblk */


getblk(x, y, bitmap)
unsigned int  x, y;                    /* coordinates in bitmap */
unsigned char     *bitmap;             /* bitmap from which to get value */
/*   return the value of the bit in the bitmap corresponding to the x,y
     coordinate provided */
{
     unsigned int  byte,               /* byte where x,y coord is located */
                   bit;                /* bit in byte where x,y is located */

     byte = indx(x, y) / (sizeof(char) * 8);
     bit = indx(x, y) % (sizeof(char) * 8);
     return((bitmap[byte] >> (7 - bit)) & 1);
}  /* getblk */
```

APPENDIX B

PERSISTENT SEARCH LISP CODE

```
;    Original implementation of persistent search in Lisp.
;
;    Written by:   Michael M. Mayer, LT, USN
;    Date:         1 September 1988
;    Version:      2.0
;    Compiler:     Written in Allegro Common Lisp on the Apple Macintosh
;    Usage:        (psearch '(startx starty) '(goalx goaly))

(require 'quickdraw)

(defvar *persistence-factor* 0.5)

(defvar *future-locs* nil)

(defvar *agenda2* nil)

(defvar *path2* nil)

(defvar *complete-trail2* nil)

;column number      0 1 2 3 4 5 6 7 8 9

(defvar *world2* '((1 1 1 1 1 1 1 1 1 1)    ;0
                   (1 0 0 0 0 1 0 0 0 1)    ;1
                   (1 0 0 0 0 1 0 0 0 1)    ;2
                   (1 0 0 1 1 1 0 0 0 1)    ;3
                   (1 0 0 0 0 1 0 1 0 1)    ;4  row number
                   (1 0 1 1 0 1 0 1 0 1)    ;5
                   (1 0 1 0 0 1 0 0 0 1)    ;6
                   (1 0 1 1 1 1 0 1 0 1)    ;7
                   (1 0 0 0 0 0 0 1 0 1)    ;8
                   (1 1 1 1 1 1 1 1 1 1 )));9

(defun init-psearch2 ()
  (setq *path2* nil)
  (setq *agenda2* nil)
  (setq *future-locs* nil)
  (setq *complete-trail2* nil)
  (setq *dead2* nil))

(defun stats2 (thewindow)
  (ask thewindow (move-to 8 214))
  (princ "The path length is: " thewindow)
  (prin1 (length *path2*) thewindow)
  (ask thewindow (move-to 8 228))
  (princ "The # of moves made is: " thewindow)
  (prin1 (length *complete-trail2*) thewindow)
  t)
```

```
(defun moves-away2 (pt1 pt2)
  (+ (abs (- (car pt1) (car pt2))) (abs (- (cadr pt1) (cadr pt2)))))

(defun update-lists (newpath)
  (let ((pathback (set-difference *path2* newpath))
        (pathforw (reverse (set-difference newpath *path2*))))
    (setq *complete-trail2*
          (append (append (reverse pathforw)
                          (remove (car *path2*) pathback))
                  *complete-trail2*))
    (draw-circles pswindow2 pathback  *white-pattern*)
    (draw-circles pswindow2 (reverse pathforw) *light-gray-pattern*)))

(defun pathlength (x y)
 (- (+ (length x) (length y)) (* 2 (length (intersection x y :test 'equal)))))

(defun compare-agenda-items (item1 item2)
  (cond ((<= (+ (get item1 'distance)
                (* (pathlength (get item1 'path) *path2*)
                   *persistence-factor*))
             (+ (get item2 'distance)
                (* (pathlength (get item2 'path) *path2*)
                   *persistence-factor*)))
         item1)
        (t item2)))

(defun best-on-agenda ()
  (do* ((agenda *agenda2* (cdr agenda))
        (best (car *agenda2*) (compare-agenda-items (car agenda) best)))
       ((null (cdr agenda)) best)))

(defun movep2 (dir loc)
  (cond ((equal dir 'north)
         (and (<= (nth (car loc) (nth (1- (cadr loc)) *world2*)) 0)
              (not(member (list (car loc) (1- (cadr loc))) *complete-trail2*
                                                           :test 'equal))
              (not(member (list (car loc) (1- (cadr loc))) *future-locs*
                                                          :test 'equal))))
        ((equal dir 'east)
         (and (<= (nth (1+ (car loc)) (nth (cadr loc) *world2*)) 0)
              (not(member (list (1+ (car loc)) (cadr loc)) *complete-trail2*
                                                           :test 'equal))
              (not(member (list (1+ (car loc)) (cadr loc)) *future-locs*
                                                          :test 'equal))))
        ((equal dir 'west)
         (and (<= (nth (1- (car loc)) (nth (cadr loc) *world2*)) 0)
              (not(member (list (1- (car loc)) (cadr loc)) *complete-trail2*
                                                          :test 'equal))
              (not(member (list (1- (car loc)) (cadr loc)) *future-locs*
                                                          :test 'equal))))
        ((equal dir 'south)
         (and (<= (nth (car loc) (nth (1+ (cadr loc)) *world2*)) 0)
              (not(member (list (car loc) (1+ (cadr loc))) *complete-trail2*
                                                          :test 'equal))
              (not(member (list (car loc) (1+ (cadr loc))) *future-locs*
                                                          :test 'equal))))))
```

```
(defun add-agenda (item)
  (setq *agenda2* (cons item *agenda2*))
  item)

(defun any-poss-dirs (start goal)
  (let ((no-bktrk nil) (temp nil))
  (when (movep2 'south start)
    (setf (get (add-agenda (gensym)) 'distance)
          (moves-away2 (setq temp (list (car start) (1+ (cadr start)))) goal))
    (setf (get (car *agenda2*) 'path) *path2*)
    (setf (get (car *agenda2*) 'loc) temp)
    (setq *future-locs* (cons temp *future-locs*))
    (setq no-bktrk t))
  (when (movep2 'west start)
    (setf (get (add-agenda (gensym)) 'distance)
          (moves-away2 (setq temp (list (1- (car start)) (cadr start))) goal))
    (setf (get (car *agenda2*) 'path) *path2*)
    (setf (get (car *agenda2*) 'loc) temp)
    (setq *future-locs* (cons temp *future-locs*))
    (setq no-bktrk t))
  (when (movep2 'east start)
    (setf (get (add-agenda (gensym)) 'distance)
          (moves-away2 (setq temp (list (1+ (car start)) (cadr start))) goal))
    (setf (get (car *agenda2*) 'path) *path2*)
    (setf (get (car *agenda2*) 'loc) temp)
    (setq *future-locs* (cons temp *future-locs*))
    (setq no-bktrk t))
  (when (movep2 'north start)
    (setf (get (add-agenda (gensym)) 'distance)
          (moves-away2 (setq temp (list (car start) (1- (cadr start)))) goal))
    (setf (get (car *agenda2*) 'path) *path2*)
    (setf (get (car *agenda2*) 'loc) temp)
    (setq *future-locs* (cons temp *future-locs*))
    (setq no-bktrk t))
  no-bktrk))

(defun psearch (start goal)
  (setq *complete-trail2* (cons start *complete-trail2*))
  (setq *path2* (cons start *path2*))
  (draw-circle2 pswindow2 start *light-gray-pattern*)
  (cond ((equal start goal) t)
        (t
         (let ((best nil) (best-loc nil) (best-path nil))
           (cond ((or (any-poss-dirs start goal) (consp *agenda2*))
                  (setq *agenda2*
                        (remove (setq best (best-on-agenda)) *agenda2*))
                  (setq best-path (get best 'path))
                  (setq best-loc (get best 'loc))
                  (setq *future-locs* (remove best-loc *future-locs*))
                  (update-lists best-path)
                  (setq *path2* best-path)
                  (psearch best-loc goal))
                 (t nil))))))
```

```
;Graphics Support

(defun showworld2 (thewindow)
  (ask thewindow (erase-rect 0 0 512 342))
  (ask thewindow (move-to 175 228))
  (prin1 *persistence-factor* thewindow)
  (do ((y 0 (1+ y)) (world *world2* (cdr world)))
      ((null world))
    (do ((x 0 (1+ x)) (row (car world) (cdr row)))
        ((null row))
      (if (equal 1 (car row))
        (ask thewindow (paint-rect (* x 20)
                                   (* y 20)
                                   (+ (* x 20) 20)
                                   (+ (* y 20) 20)))))))

(defun draw-circles (thewindow ptlist pattern)
  (cond ((null ptlist) t)
        (t
         (draw-circles thewindow (cdr ptlist) pattern)
         (draw-circle2 thewindow (car ptlist) pattern))))

(defun draw-circle2 (thewindow pt pattern)
  (prog ()
    no-shift-key
    (cond ((shift-key-p) (return))
          (t (go no-shift-key))))
  (ask thewindow (fill-oval pattern
                            (* (car pt) 20)
                            (* (cadr pt) 20)
                            (+ (* (car pt) 20) 20)
                            (+ (* (cadr pt) 20) 20)))
  (ask thewindow (frame-oval (* (car pt) 20)
                             (* (cadr pt) 20)
                             (+ (* (car pt) 20) 20)
                             (+ (* (cadr pt) 20) 20))))


;Windows

(setq pswindow2 (onecf *window*
                       :window-title "Persistent Search"
                       :window-type :tool
                       :window-position #@ (25 60)
                       :window-size #@ (200 232)
                       :window-font '("New York" 10)))
```

```
;Menus

(setq psmenu1 (oneof *menu* :menu-title "PS Actions"))
(ask psmenu1 (add-menu-items
              (oneof *menu-item*
                     :menu-item-title "Initialize"
                     :menu-item-action '(init-psearch2))
              (oneof *menu-item*
                     :menu-item-title "Show World"
                     :menu-item-action '(showworld2 pswindow2))
              (oneof *menu-item*
                     :menu-item-title "Statistics"
                     :menu-item-action '(stats2 pswindow2))
              (oneof *menu-item*
                     :menu-item-title "Run (1 1) (8 8)"
                     :menu-item-action '(eval-enqueue
                                         '(psearch '(1 1) '(8 8))))
              (oneof *menu-item*
                     :menu-item-title "Run (6 1) (8 8)"
                     :menu-item-action '(eval-enqueue
                                         '(psearch '(6 1) '(8 8))))
              (oneof *menu-item*
                     :menu-item-title "Run (1 4) (8 4)"
                     :menu-item-action '(eval-enqueue
                                         '(psearch '(1 4) '(8 4))))))

(ask psmenu1 (menu-install))

(setq psmenu2 (oneof *menu* :menu-title "Persistence"))
(ask psmenu2 (add-menu-items
              (oneof *menu-item*
                     :menu-item-title "0.0"
                     :menu-item-action '(setq *persistence-factor* 0.0))
              (oneof *menu-item*
                     :menu-item-title "0.2"
                     :menu-item-action '(setq *persistence-factor* 0.2))
              (oneof *menu-item*
                     :menu-item-title "0.3"
                     :menu-item-action '(setq *persistence-factor* 0.3))
              (oneof *menu-item*
                     :menu-item-title "0.41"
                     :menu-item-action '(setq *persistence-factor* 0.41))
              (oneof *menu-item*
                     :menu-item-title "0.5"
                     :menu-item-action '(setq *persistence-factor* 0.5))
              (oneof *menu-item*
                     :menu-item-title "0.66"
                     :menu-item-action '(setq *persistence-factor* 0.66))
              (oneof *menu-item*
                     :menu-item-title "0.8"
                     :menu-item-action '(setq *persistence-factor* 0.8))
              (oneof *menu-item*
                     :menu-item-title "1.0"
                     :menu-item-action '(setq *persistence-factor* 1.0))))

(ask psmenu2 (menu-install))
```

APPENDIX C

EMPIRICAL TESTING C CODE

```
makemaze()                          /* generate a random maze */
{
     unsigned int  x, y,            /* node coordinates */
               i,                   /* loop index */
               stepclear,           /* movement is clear in current direction */
               dir = 0,             /* current direction */
               lastdir = 0,         /* last direction */
               mazestart;           /* starting point of maze creation */
     int       ptop = 0;            /* top of the path stack */
     float     getrandom();         /* returns random number in range specified */

     for (i = 0; i < size; i++) {    /* set border of maze */
          setblk(0, i, world);
          setblk(size - 1, i, world);
          setblk(i, 0, world);
          setblk(i, size - 1, world);
     }
          /* picks a starting point for maze creation */
     mazestart = random() % ((size - 2) * (size - 2)) + size + 1;
     x = getx(mazestart);
     y = gety(mazestart);
     setblk(x, y, trail);
     pstack[ptop] = mazestart;

     while(TRUE) {
          if(getrandom(1.0) < density)    /* if rand less than density */
          {
               dir = random() % 4;        /* change direction */
               switch(lastdir)            /* block previous direction */
               {
                    case 0:
                         if(!getblk(x, y - 1, trail))
                              setblk(x, y - 1, world);
                         break;
                    case 1:
                         if(!getblk(x + 1, y, trail))
                              setblk(x + 1, y, world);
                         break;
                    case 2:
                         if(!getblk(x, y + 1, trail))
                              setblk(x, y + 1, world);
                         break;
                    case 3:
                         if(!getblk(x - 1, y, trail))
                              setblk(x - 1, y, world);
                         break;
               }  /* switch(lastdir) */
          }  /* if */
```

```
            stepclear = FALSE;

            switch(dir)       /* see if node in current direction is clear */
            {
                 case 0:
                      if(!getblk(x, y - 1, trail) && !getblk(x, y - 1, world)
#ifdef TREE
                      && treetest(x, y - 1)  /* make maze a tree */
#endif
                 )
                      {
                           stepclear = TRUE;
                           y--;
                      }
                      break;
                 case 1:
                      if(!getblk(x + 1, y, trail) && !getblk(x + 1, y, world)
#ifdef TREE
                      && treetest(x + 1, y)  /* make maze a tree */
#endif
                 )
                      {
                           stepclear = TRUE;
                           x++;
                      }
                      break;
                 case 2:
                      if(!getblk(x, y + 1, trail) && !getblk(x, y + 1, world)
#ifdef TREE
                      && treetest(x, y + 1)  /* make maze a tree */
#endif
                 )
                      {
                           stepclear = TRUE;
                           y++;
                      }
                      break;
                 case 3:
                      if(!getblk(x - 1, y, trail) && !getblk(x - 1, y, world)
#ifdef TREE
                      && treetest(x - 1, y)  /* make maze a tree */
#endif
            )
                      {
                           stepclear = TRUE;
                           x--;
                      }
                      break;
            } /* switch(dir) */

            if(!stepclear)    /* if current path not clear */
            {
                 int  count = 0,    /* number of clear moves */
                      choice,       /* which clear move chosen */
                      dirs[4];      /* array of up to four clear moves */
```

```
            while(!count)     /* find all clear moves */
            {
                if(!getblk(x, y - 1, trail) && !getblk(x, y - 1, world)
#ifdef TREE
                && treetest(x, y - 1)  /* make maze a tree */
#endif
                )
                    dirs[count++] = 0;
                if(!getblk(x + 1, y, trail) && !getblk(x + 1, y, world)
#ifdef TREE
                && treetest(x + 1, y)  /* make maze a tree */
#endif
                )
                    dirs[count++] = 1;
                if(!getblk(x, y + 1, trail) && !getblk(x, y + 1, world)
#ifdef TREE
                && treetest(x, y + 1)  /* make maze a tree */
#endif
                )
                    dirs[count++] = 2;
                if(!getblk(x - 1, y, trail) && !getblk(x - 1, y, world)
#ifdef TREE
                && treetest(x - 1, y)  /* make maze a tree */
#endif
                )
                    dirs[count++] = 3;
                if(!count)              /* if no clear moves */
                {
                    ptop--;             /* pop path node off stack */
                    if(ptop == -1)      /* if stack empty, we're done!*/
                    {
                        setstartandgoal();  /* get random start and goal */
                        return;
                    }
                    x = getx(pstack[ptop]);
                    y = gety(pstack[ptop]);
                }
                else     /* pick one of the clear moves */
                {
                    choice = random() % count;
                    dir = dirs[choice];
                    switch(dir)
                    {
                        case 0: y--; break;
                        case 1: x++; break;
                        case 2: y++; break;
                        case 3: x--; break;
                    }  /* switch */
                } /* if */
            }  /* while */
        }  /* if */
        pstack[++ptop] = indx(x, y);    /* put node on path stack */
        setblk(x, y, trail);            /* add to trail bitmap */
        lastdir = dir;
    }  /* while */
}  /* makemaze */
```

```
#ifdef TREE

treetest(x, y)          /* see if adding node to maze would cause a cycle */
unsigned int  x, y;     /* coordinates of node to be tested */
{
     int connect = 0;  /* number of clear adjacent nodes in maze */

     if(getblk(x, y - 1, trail))
          connect++;
     if(getblk(x + 1, y, trail))
          connect++;
     if(getblk(x, y + 1, trail))
          connect++;
     if(getblk(x - 1, y, trail))
          connect++;
     if(connect > 1)
     {
          setblk(x, y, world);
          return FALSE;
     }
     else
          return TRUE;
}  /* treetest */

#endif


setstartandgoal() /* finds an open space to place the start and goal nodes */
{
     do
     {
          Start = random() % ((size - 2) * (size - 2)) + size + 1;
     } while(!getblk(getx(Start), gety(Start), trail));
     do
     {
          goal = random() % ((size - 2) * (size - 2)) + size + 1;
     } while(!getblk(getx(goal), gety(goal), trail));
}  /* setgoal */


float getrandom(range)
float range;                            /* value returned is in 0 - range */
/* returns random floating point value in the range 0 - range */
{
     return((float)random() / (float)0x7FFFFFFF * range);
}  /* getrandom */
```

```
#ifdef PM

print_maze()
{
     unsigned int i, j;          /* loop variables */

     for(i = 0; i < size; i++)   /* print maze */
     {
          for(j = 0; j < size; j++)
               if(getblk(j, i, world))
                    printf(" *");
               else
                    printf("  ");

          printf("\n");
     }
     for(i = 0; i < size; i++)   /* print maze and trail */
     {
          for(j = 0; j < size; j++)  /* shows blocked open space */
          {
               if(getblk(j, i, world) && getblk(j, i, trail))
                    fprintf(stderr, "trail/world mismatch: %d, %d\n", j, i);
               if(getblk(j, i, world))
                    printf(" X");
               else if(getblk(j, i, trail))
                    printf(" O");
               else
                    printf("  ");
          }
          printf("\n");
     }
}  /* print_maze */

#else

print_maze()
{

}  /* print_maze */

#endif
```

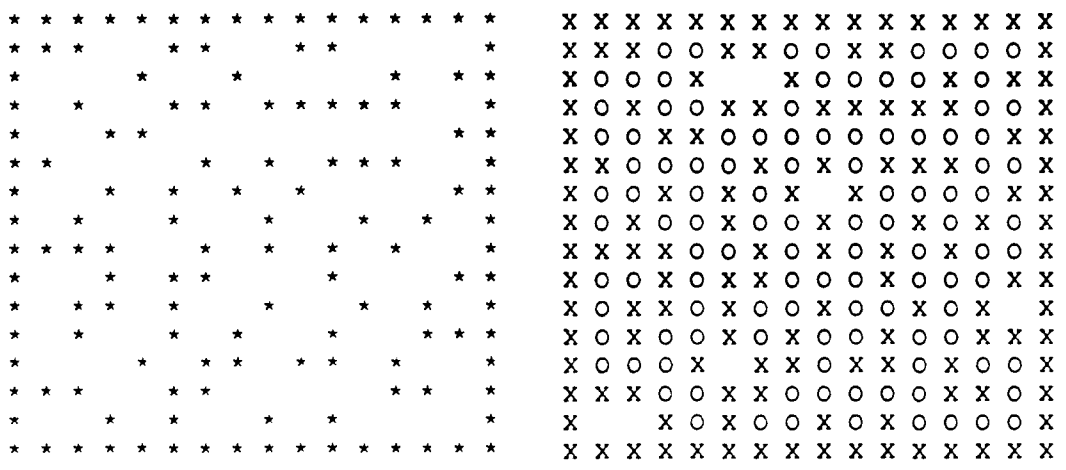
```
dfs()    /* depth-first search (hill-climbing) */
{
     unsigned char      *dfstrail;
     unsigned int       *stack;

     unsigned int       x, y,          /* coordinates of current node */
                        gx, gy,        /* coordinates of goal node */
                        count = 0,     /* number of successor nodes */
                        choice,        /* successor node chosen */
                        dirs[4];       /* array of successor nodes */

     int                top = 0;       /* top of stack pointer for stack */

          /* get coordinates */
     x = getx(Start);
     y = gety(Start);
     gx = getx(goal);
     gy = gety(goal);

          /* allocate dynamic memory for the stack and dfs path trail */
     stack = (unsigned int *)calloc(size * size, sizeof(unsigned int));
     dfstrail = (unsigned char *)calloc(size * size / (sizeof(char) * 8),
          sizeof(char));

          /* make start node current node and mark the trail */
     setblk(x, y, dfstrail);
     stack[top] = Start;

     while(stack[top] != goal)       /* while we are not at the goal */
     {
          count = 0;
          while(count == 0)          /* while we have no successors */
          {
#ifdef PS
               /* print path of dfs search */
               printf("DFS visited: %d, %d\n", x, y);
#endif
               /* find legal successors */
               if(!getblk(x, y - 1, dfstrail) && !getblk(x, y - 1, world))
                    dirs[count++] = indx(x, y - 1);
               if(!getblk(x + 1, y, dfstrail) && !getblk(x + 1, y, world))
                    dirs[count++] = indx(x + 1, y);
               if(!getblk(x, y + 1, dfstrail) && !getblk(x, y + 1, world))
                    dirs[count++] = indx(x, y + 1);
               if(!getblk(x - 1, y, dfstrail) && !getblk(x - 1, y, world))
                    dirs[count++] = indx(x - 1, y);

               if(count == 0)     /* if no successors */
               {
                    top--;        /* pop stack */
                    if(top == -1)      /* if stack is empty */
                    {
                         free(stack);       /* free dynamic memory */
                         free(dfstrail);
                         printf("DFS: No solution possible!\n");
                         return;            /* search failed, return */
                    }
```

```
                x = getx(stack[top]);
                y = gety(stack[top]);
            }
            else
            {
                    /* pick a direction */
                choice = closest(dirs, count, gx, gy);
                x = getx(choice);
                y = gety(choice);
            } /* if */
                /* keep statistics */
            dfsmoves++;
            dfscompares++;
        }  /* while */
        stack[++top] = indx(x, y);      /* push stack */
        setblk(x, y, dfstrail);         /* mark trail */
    }  /* while */
#ifdef PP
    printf("DFS visited: %d, %d\n", x, y);
#endif
    free(stack);       /* free dynamic memory */
    free(dfstrail);
}  /* dfs */


int closest(dirs, count, gx, gy)
unsigned int  dirs[],           /* array of node indexes */
              count,            /* number of indexes */
              gx, gy;           /* coordinates of the goal */
/*   closest returns the index of the node which is the closest to the goal.
*    It chooses from the up to four indexes contained in dirs.  count gives
*    the number of indexes contained in dirs.  The indexes contained in dirs
*    are the clear nodes surrounding the current node.  gx and gy
*    are the x and y coordinates of the goal */
{
    int i,                      /* loop variable */
        winner = dirs[0];       /* index of best node up till then */

    for(i = 1; i < count; i++)
        if(dist(getx(dirs[i]), gety(dirs[i]), gx, gy) <
           dist(getx(winner), gety(winner), gx, gy))
            winner = dirs[i];
    return winner;
}  /* closest */
```

APPENDIX D

RESULTS OF SAMPLE TEST CASE

Maze density: 0.50



Persistance factor: 0.90

ps:23 moves **ps:125 comparisons**

PS visited: 6, 7	PS visited: 7, 7	PS visited: 7, 8
PS visited: 7, 9	PS visited: 8, 9	PS visited: 9, 9
PS visited: 9, 8	PS visited: 9, 9	PS visited: 9, 10
PS visited: 10, 10	PS visited: 9, 10	PS visited: 9, 11
PS visited: 8, 11	PS visited: 8, 12	PS visited: 8, 13
PS visited: 9, 13	PS visited: 10, 13	PS visited: 11, 13
PS visited: 11, 12	PS visited: 11, 11	PS visited: 12, 11
PS visited: 12, 10	PS visited: 12, 9	PS visited: 13, 9

dfs: 163	moves	dfs: 163	comparisons
DFS visited: 6, 7	DFS visited: 7, 7		DFS visited: 7, 8
DFS visited: 7, 9	DFS visited: 8, 9		DFS visited: 9, 9
DFS visited: 9, 8	DFS visited: 9, 7		DFS visited: 10, 7
DFS visited: 10, 6	DFS visited: 11, 6		DFS visited: 12, 6
DFS visited: 13, 6	DFS visited: 13, 5		DFS visited: 13, 4
DFS visited: 13, 3	DFS visited: 13, 2		DFS visited: 13, 1
DFS visited: 14, 1	DFS visited: 13, 1		DFS visited: 12, 1
DFS visited: 11, 1	DFS visited: 11, 2		DFS visited: 10, 2
DFS visited: 9, 2	DFS visited: 8, 2		DFS visited: 8, 1
DFS visited: 7, 1	DFS visited: 8, 1		DFS visited: 8, 2
DFS visited: 9, 2	DFS visited: 10, 2		DFS visited: 11, 2
DFS visited: 11, 1	DFS visited: 12, 1		DFS visited: 13, 1
DFS visited: 13, 2	DFS visited: 13, 3		DFS visited: 14, 3
DFS visited: 13, 3	DFS visited: 13, 4		DFS visited: 12, 4
DFS visited: 11, 4	DFS visited: 10, 4		DFS visited: 9, 4
DFS visited: 9, 5	DFS visited: 9, 4		DFS visited: 8, 4
DFS visited: 7, 4	DFS visited: 7, 5		DFS visited: 7, 4
DFS visited: 7, 3	DFS visited: 7, 4		DFS visited: 6, 4
DFS visited: 5, 4	DFS visited: 5, 5		DFS visited: 4, 5
DFS visited: 4, 6	DFS visited: 4, 7		DFS visited: 4, 8
DFS visited: 5, 8	DFS visited: 4, 8		DFS visited: 4, 9
DFS visited: 4, 10	DFS visited: 4, 11		DFS visited: 3, 11
DFS visited: 3, 12	DFS visited: 3, 13		DFS visited: 4, 13
DFS visited: 4, 14	DFS visited: 4, 13		DFS visited: 3, 13
DFS visited: 3, 12	DFS visited: 2, 12		DFS visited: 1, 12
DFS visited: 1, 11	DFS visited: 1, 10		DFS visited: 1, 9
DFS visited: 2, 9	DFS visited: 1, 9		DFS visited: 1, 10
DFS visited: 1, 11	DFS visited: 1, 12		DFS visited: 2, 12
DFS visited: 3, 12	DFS visited: 3, 11		DFS visited: 4, 11
DFS visited: 4, 10	DFS visited: 4, 9		DFS visited: 4, 8
DFS visited: 4, 7	DFS visited: 3, 7		DFS visited: 4, 7
DFS visited: 4, 6	DFS visited: 4, 5		DFS visited: 3, 5
DFS visited: 2, 5	DFS visited: 2, 6		DFS visited: 1, 6
DFS visited: 1, 7	DFS visited: 1, 6		DFS visited: 2, 6
DFS visited: 2, 5	DFS visited: 2, 4		DFS visited: 1, 4
DFS visited: 1, 3	DFS visited: 1, 2		DFS visited: 2, 2
DFS visited: 3, 2	DFS visited: 3, 3		DFS visited: 4, 3
DFS visited: 3, 3	DFS visited: 3, 2		DFS visited: 3, 1
DFS visited: 4, 1	DFS visited: 3, 1		DFS visited: 3, 2
DFS visited: 2, 2	DFS visited: 1, 2		DFS visited: 1, 3
DFS visited: 1, 4	DFS visited: 2, 4		DFS visited: 2, 5
DFS visited: 3, 5	DFS visited: 4, 5		DFS visited: 5, 5
DFS visited: 5, 4	DFS visited: 6, 4		DFS visited: 7, 4
DFS visited: 8, 4	DFS visited: 9, 4		DFS visited: 10, 4
DFS visited: 11, 4	DFS visited: 12, 4		DFS visited: 13, 4
DFS visited: 13, 5	DFS visited: 14, 5		DFS visited: 13, 5
DFS visited: 13, 6	DFS visited: 12, 6		DFS visited: 12, 7
DFS visited: 12, 6	DFS visited: 11, 6		DFS visited: 10, 6
DFS visited: 10, 7	DFS visited: 9, 7		DFS visited: 9, 8
DFS visited: 9, 9	DFS visited: 9, 10		DFS visited: 10, 10
DFS visited: 9, 10	DFS visited: 9, 11		DFS visited: 8, 11
DFS visited: 8, 12	DFS visited: 8, 13		DFS visited: 9, 13
DFS visited: 10, 13	DFS visited: 11, 13		DFS visited: 11, 12
DFS visited: 11, 11	DFS visited: 12, 11		DFS visited: 12, 10
DFS visited: 12, 9	DFS visited: 13, 9		

LIST OF REFERENCES

Brassard, Gilles and Bratley, Paul, *Algorithmics Theory and Practice*, Prentice Hall, 1988.

Hart, P.E., Nilsson, N.J., and Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems, Science, and Cybernetics*, Vol. 4, No. 2, 1968.

Hu, T.C. and Wachs, Michelle L., "Binary Search on a Tape," *SIAM Journal on Computing*, Vol. 16, No. 3, June 1987.

Rowe, Neil C., *Artificial Intelligence Through Prolog*, Prentice Hall, 1988.

Kumar, V., "Branch-and-bound Search,", *Encyclopedia of Artificial Intelligence*, John Wiley and Sons, 1987.

BIBLIOGRAPHY

Chattery, R., "Some Heuristics for the Navigation of a Robot," *The International Journal of Robotics Research*, Vol. 4, No. 1, Spring 1985.

Hu, T.C. and Shing, M.T., "The a-ß Routing," *VLSI: Circuit Layout Theory*, IEEE Press, 1985.

Goodpasture, Richard Paul, *A Computer Simulation Study of an Expert System for Walking Machine Motion Planning*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1987.

Naval Postgraduate School Report NPS52-86-021, *Solving Global Two Dimensional Routing Problems Using Snell's Law and A* Search*, by Richbourg, R.F, Rowe, Neil C., Zyda, Michael J., and McGhee, Robert B., October 1986.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Superintentent Attn: Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Naval Security Group Activity Skaggs Island Attn: LT Michael M. Mayer Box 1099 NSGA Skaggs Island Sonoma, California 95476	4
4.	Prof. Man-Tak Shing, Code 52Sh Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	16
5.	Prof. Robert B. McGhee, Code 52Mz Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
6.	Prof. Gordon H. Bradley, Code 55Bz Department of Operations Research Naval Postgraduate School Monterey, California 93943-5000	1
7.	Prof. Neil C. Rowe, Code 52Rp Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1